版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

仅供非商业用途或交流学习使用

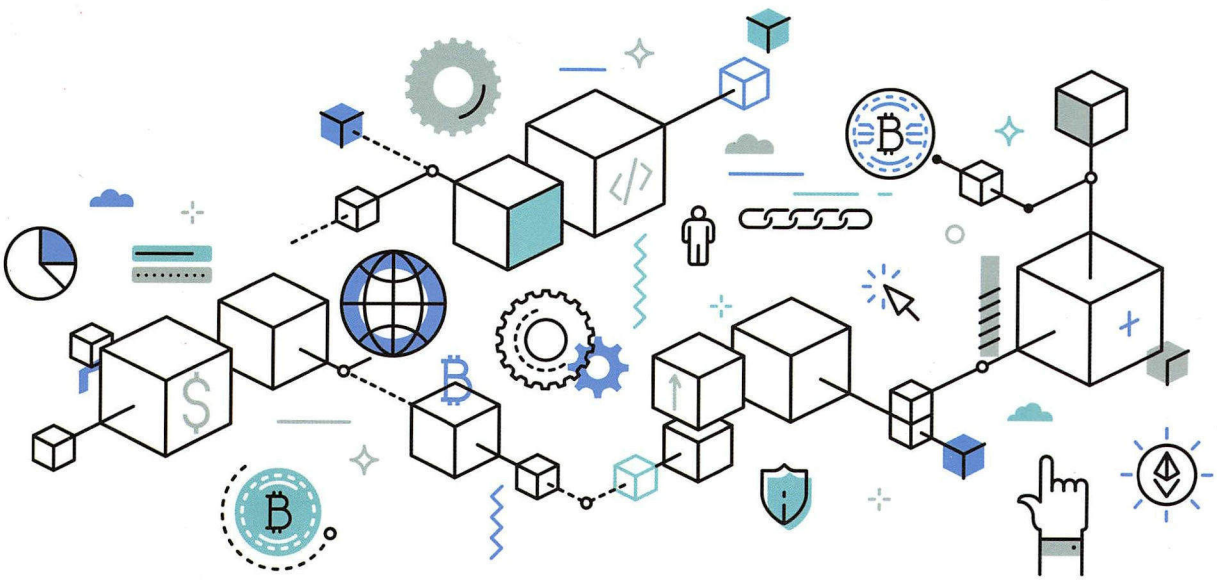
涵盖Fabric基础技术和原理　Fabric技术与实践紧密结合
多角度阐述和分析十大经典案例

区块链技术丛书

区块链网络构建和应用

基于超级账本Fabric的商业实践

陆平　张晗　张再军　田江磊　◎等编著



机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

本书特点

Fabric 是当前联盟链广泛采用的开源技术。本书对从事区块链技术研究、应用开发的单位和个人有较大的助益！同时可以促进我国区块链技术的全面发展！

对 Fabric 有全面、深入的讲解，从原理和工程的不同角度进行了剖析，对有志于 Fabric 开发实践的读者有很强的借鉴意义。

对区块链数据库选用方案、私钥证书管理方案、数据上链方案、背书验证方案做了专项描述，这些方案取材于区块链应用实践中的宝贵经验，值得分享给各位读者。

精选的区块链案例进行剖析，让读者对区块链的了解不仅仅停留在理论和操作层面，而是将理论和实践相结合，透过现象看本质，最终使读者形成对区块链的多维度认知。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

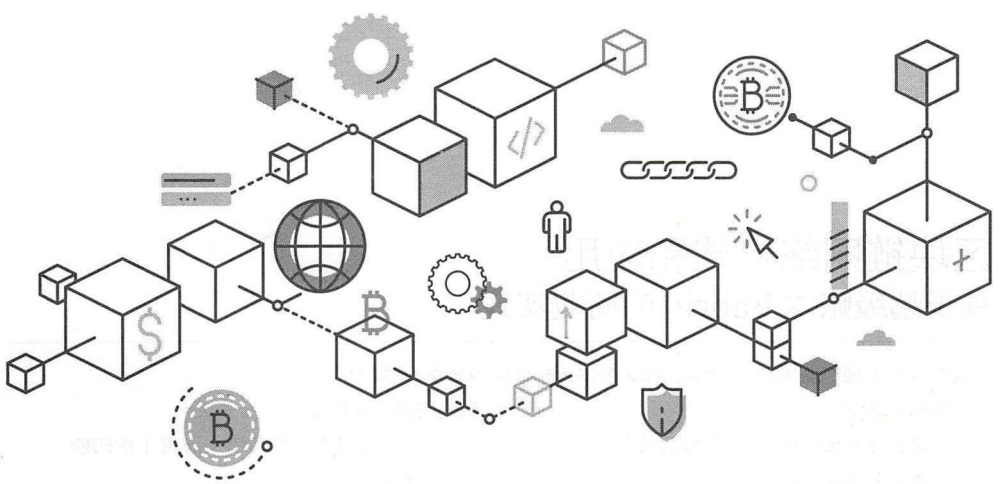
仅供非商业用途或交流学习使用

区块链
技术丛书

区块链网络构建和应用

基于超级账本Fabric的商业实践

陆平 张晗 张再军 田江磊 钱煜明 戚晨 ◎ 编著



机械工业出版社
China Machine Press

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

区块链网络构建和应用：基于超级账本 Fabric 的商业实践 / 陆平等编著 . —北京：机械工业出版社，2018.9
（区块链技术丛书）

ISBN 978-7-111-60911-7

I. 区… II. 陆… III. 电子商务 – 支付方式 – 研究 IV. F713.361.3

中国版本图书馆 CIP 数据核字（2018）第 210926 号

区块链网络构建和应用
基于超级账本 Fabric 的商业实践

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：陈佳媛　　责任校对：李秋荣
印　　刷：北京诚信伟业印刷有限公司　　版　　次：2018 年 9 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：22
书　　号：ISBN 978-7-111-60911-7　　定　　价：79.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 • 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

Foreword 推 荐 序

区块链技术与比特币是“孪生姐妹”，今年是它们的10周年华诞。作为首个完全去中心化的公众链加密数字货币——比特币的底层技术是互联网诞生以来最大的一次技术革命，它将对人类社会产生全方位的冲击，包括人类经济、政治、社会的各个领域。

回顾一下互联网的发展所经历的若干阶段，从最初只是职业人士用于科研圈的文件信息交流网，到发明Web技术用于老百姓的电子商务及门户网站消费型网络，再后来由人人互联到人物或物物互联的万物互联物联网，近年正向价值制造和价值创造的生产型网络——工业互联网进化；可以看出网络技术确实成为人类最重要的社会生活和生产基础实施，但是到目前为止它们的一个共同特点是网络运营或管理模式都需要一个中心化的机构来管理运营，所以它们的重要共同特点都是中心控制的网络。

中心化的缺陷是它需要专门的中心控制管理中心所产生的成本；它使得行业的垄断骤然产生达到赢者通吃的地步；中心机构滥用了用户的数据或隐私信息。区块链技术正是构建去中心化不需要第三方的，很低成本或不需成本的信任关系，故是传递价值的互联网。

本书有广度也有深度，有全景理论介绍更有各个行业的应用案例。本书从区块链基础，通俗讲解区块链技术原理和网络构造，完整描述区块链的实际案例，有严谨的推导，有原创设计，也有对区块链的新的技术和挑战方面的研讨。作者从产业的角度，非常重视区块链技术的发展，并在行业标准制定、关键技术研究、产品研发、应用创新上投入了相当多的资源，成功研制开发出自主知识产权的区块链平台产品，并在政府、企业不同的领域进行了创新应用的研讨。

相信本书对有志从事区块链领域的研究和应用的入门者来说，是一本绝佳的入门书籍。即便是你有了一定的基础，对于相关领域的决策者、从业者、学术研究者来说，本书也是极具新意和实用价值的参考资料。

北京大学南燕区块链技术实验室

中国计算机学会CCF区块链专委会　李晖

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

前　言 Preface

区块链技术首先在比特币上得到应用并受到广泛关注。目前区块链技术作为去中心化记账（Decentralized Ledger Technology，DLT）平台的核心技术，被认为在金融、政务、征信、物联网、经济贸易结算、资产管理等众多领域都拥有广泛的应用前景。区块链技术自身尚处于快速发展的初级阶段，现有区块链系统在设计和实现中利用了分布式系统、密码学、博弈论、网络协议等诸多学科的知识，为学习原理和实践应用都带来了不小的挑战。区块链的概念已经完全超出了数字货币领域，在社会的各行各业都获得越来越多的关注。业界对区块链的技术报以极大的热情。

目前市面上有很多有关区块链的书籍，大多只对技术的起源、原理、发展、应用场景以及未来的趋势进行了介绍，往往仅停留在理论层面，缺少真正接地气的案例实践的支撑。

本书对精选的区块链案例实践进行剖析，让读者对区块链的了解不仅仅停留在理论层面，而是理论和实践相结合，透过现象看本质，最终形成读者自己对区块链的一个全新认识。

区块链领域涉及的技术很宽泛，发展速度也很快，多项技术是相辅相成的。本书在介绍区块链技术原理部分的同时，也将其关联的分布式技术、密码学技术、人工智能等介绍给读者。结合区块链的开源项目剖析和实践，引导读者自行搭建区块链网络，加深对区块链的理解。

在介绍区块链应用实践的时候，重点从工程的角度完整地描述一个区块链项目的背景、需求、项目方案和部署、项目的功能设计、接口设计、流程设计。并从项目运营优化的角度提出了智能合约的设计和可视化二次开发、项目的可视化运营和部署的方案。通过对不同领域的一些区块链典型案例的剖析，让读者从区块链技术到项目落地有一个全新的认识。

本书的最后一章研讨了区块链和大数据的关系、区块链和人工智能的关系、BCaaS、欧盟的“通用数据保护条例”（CDPR）对区块链的影响、区块链发展中面临的挑战，让读者对区块链有了更加全面的认识。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

本书由 8 章组成。

第 1 章介绍了区块链基础，包含了区块链领域的基础概念术语、核心技术、热门区块链平台等内容。第 2 章对分布式系统技术进行了介绍，区块链首先是一个分布式系统，了解区块链离不开分布式系统技术。第 3 章介绍了密码学安全技术。公私钥密码算法是区块链系统的基石。区块链之所以被称为信任的机器，其中的密码安全技术是重要的一个环节。第 4 章以 Fabric 开源项目为基础，引导用户构建一个自己的区块链网络。第 5 章基于开源项目源码的分析，帮助用户深入了解区块链账本、共识算法、加密等核心技术的实现。第 6 章通过介绍区块链在政务服务数据共享及服务商的项目实践，让读者对区块链项目落地有深入的了解。第 7 章对区块链在各个行业的典型应用进行了介绍，让读者对区块链应用实践有更加全面的了解。第 8 章对区块链未来发展进行了展望，针对区块链与其他技术的融合、区块链技术发展中面临的挑战，从性能、安全等多维度进行了研讨。综合本书的内容，全书可分为理论和实践两部分，前 3 章注重理论，后 5 章注重实践，图文并茂，内容丰富，由浅入深，讲解全面，具有很强的借鉴性。

作者在区块链技术领域有多年的技术和应用实践经验。本书结合区块链最新技术趋势和作者的长期实践，对区块链技术提出系统的理解，对区块链项目实践提供了思路和建议。本书探索区块链概念的来龙去脉，剥茧抽丝，剖析关键技术原理，同时讲解实践应用。在区块链项目开发和落地的过程中，作者将累积的一些实践经验也通过本书一并分享出来，希望能推动区块链技术的早日成熟，出现更多的应用场景。

我们在编写过程中，通过网络搜索工具 baidu 或者 google 搜索和查阅了一些文章，并引用了部分文字，有些难以确定具体出处，无法一一列举。如读者发现未标明出处的引用，可以通过出版社告知作者，在本书的下个版本中会补充到参考文献中或做其他修订处理。在书中以及本书描述的产品中，出现的商标、产品名称、服务名称以及公司名称由各自的所有者拥有，本书内容不构成任何形式的承诺，除非适法要求，作者及出版社对本书所有内容不提供任何明示或暗示的保证。

在法律允许的范围内，本书作者及出版社在任何情况下都不对因使用本书相关内容而产生任何特殊的、附带的、间接的、继发性的损害承担责任，也不对任何利润、数据、商誉或预期的损失进行赔偿。

由于作者水平有限，书中难免存在一些错误和不足之处，敬请读者批评指正。非常感谢在百忙之中为本书作序的李挥教授，同时本书的撰写得到很多领导和同事的大力支持，在此一并表示谢意。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

目　录 *Contents*

推荐序
前　言

第 1 章　区块链基础 …… 1

1.1　区块链常用名词解释 …… 2
1.2　区块链的发展历程 …… 4
1.3　区块链概念 …… 7
1.3.1　区块链是什么 …… 7
1.3.2　区块链的特性 …… 7
1.3.3　区块链分类 …… 8
1.3.4　区块链构建信任 …… 9
1.3.5　区块链的社会价值 …… 10
1.4　区块链核心技术 …… 10
1.4.1　综述 …… 10
1.4.2　区块链结构 …… 15
1.4.3　智能合约 …… 17
1.4.4　跨链技术 …… 20
1.4.5　ILP 详解及应用 …… 26
1.5　热门区块链平台对比分析 …… 31
1.5.1　分析背景 …… 31
1.5.2　平台简介 …… 31
1.5.3　类别对比 …… 33
1.5.4　共识机制对比 …… 34
1.5.5　性能对比 …… 35
1.5.6　隐私保护对比 …… 36
1.5.7　智能合约对比 …… 37
1.5.8　技术路线对比 …… 37
1.5.9　经济模型对比 …… 38

第 2 章　分布式系统技术 …… 41

2.1　一致性问题 …… 41
2.1.1　问题挑战 …… 42
2.1.2　一致性的要求 …… 42
2.1.3　一致性模型 …… 43
2.2　一致性的共识算法 …… 45
2.2.1　问题挑战 …… 45
2.2.2　常见算法 …… 45
2.2.3　理论界限 …… 48
2.3　FlP 不可能原理 …… 49
2.4　CAP 原理 …… 49
2.4.1　CAP 原理定义 …… 49
2.4.2　应用场景 …… 50
2.5　ACID 原则 …… 51
2.6　可靠性指标 …… 52

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2.7 小结 …… 53

第 3 章 密码学安全技术 …… 54

3.1 Hash 算法与数字摘要 …… 54
3.1.1 Hash 定义 …… 55
3.1.2 常见算法 …… 55
3.1.3 性能 …… 56
3.1.4 数字摘要 …… 56
3.1.5 Hash 攻击与防护 …… 56
3.1.6 区块链中的 Hash 应用 …… 57
3.2 加密算法 …… 57
3.2.1 加解密系统基本组成 …… 57
3.2.2 对称加密算法 …… 58
3.2.3 非对称加密算法 …… 59
3.2.4 选择明文攻击 …… 60
3.2.5 混合加密机制 …… 60
3.2.6 离散对数与 DH 密钥交换协议 … 61
3.2.7 区块链加密技术 …… 62
3.3 消息认证码与数字签名 …… 64
3.3.1 消息认证码 …… 64
3.3.2 数字签名 …… 64
3.3.3 安全性 …… 65
3.3.4 区块链数字签名 …… 65
3.4 数字证书 …… 66
3.4.1 X.509 证书规范 …… 66
3.4.2 证书格式 …… 67
3.4.3 证书信任链 …… 68
3.5 PKI 体系 …… 69
3.5.1 PKI 基本组件 …… 69
3.5.2 证书的签发 …… 69
3.5.3 证书的撤销 …… 71
3.6 Merkle 树结构 …… 71
3.6.1 快速对比大量数据 …… 72
3.6.2 快速定位修改 …… 72
3.6.3 零知识证明 …… 72
3.7 布隆过滤器 …… 72
3.7.1 基于 Hash 值的快速查找 …… 73
3.7.2 更高效的布隆过滤器 …… 73
3.8 同态加密 …… 73
3.8.1 定义 …… 73
3.8.2 问题与挑战 …… 74
3.8.3 函数加密 …… 75
3.9 其他问题 …… 75
3.9.1 零知识证明概述 …… 75
3.9.2 量子密码学 …… 75
3.9.3 社交工程学 …… 76
3.9.4 安全多方计算 …… 76
3.10 小结 …… 76

第 4 章 构建 Fabric 区块链网络 …… 78

4.1 超级账本 Fabric 简介 …… 78
4.2 Fabric 特性和架构设计 …… 80
4.2.1 Fabric 特性 …… 80
4.2.2 Fabric 系统架构 …… 82
4.3 Fabric 部署 …… 85
4.3.1 单节点部署 …… 85
4.3.2 多节点区块链网络部署 …… 90
4.4 Fabric 开发 …… 97
4.4.1 ChainCode 开发 …… 97
4.4.2 应用开发示例 …… 117
4.5 Fabric 方案设计 …… 125
4.5.1 数据库选用方案 …… 125

4.5.2 私钥证书管理方案 …… 127
4.5.3 数据上链方案 …… 132
4.5.4 背书验证方案 …… 133

第 5 章 Fabric 源代码解析 …… 135

5.1 概述 …… 135
5.1.1 源码中的简拼 …… 136
5.1.2 源码中的惯例 …… 137
5.1.3 源码目录的基本结构 …… 138
5.2 peer 命令结构 …… 138
5.2.1 peer 目录结构 …… 138
5.2.2 第三方包 …… 139
5.2.3 peer 命令结构解析 …… 140
5.2.4 子命令结构解析 …… 140
5.3 日志系统 …… 142
5.3.1 go-logging 简介 …… 142
5.3.2 flogging …… 142
5.4 配置系统 …… 143
5.4.1 viper 简介 …… 143
5.4.2 viper 搜索路径和文件 …… 144
5.4.3 InitViper …… 144
5.4.4 安全文件配置 …… 145
5.4.5 命令选项配置 …… 145
5.4.6 环境变量配置 …… 146
5.5 账本 …… 146
5.5.1 账本简介 …… 146
5.5.2 数据存储服务对象 …… 149
5.5.3 四类账本 …… 151
5.6 加密服务 …… 171
5.6.1 BCCSP 的接口和选项 …… 172
5.6.2 SW 实现方式 …… 174
5.6.3 PKCS11 实现方式 …… 177
5.6.4 BCCSP 工厂 …… 179
5.7 chaincode …… 180
5.7.1 chaincode 元数据 …… 180
5.7.2 chaincode 元工具 …… 184
5.7.3 SCC 的注册和部署 …… 185
5.7.4 ACC 的安装和部署 …… 190
5.8 Orderer 服务 …… 199
5.8.1 简介 …… 199
5.8.2 模块 …… 200
5.8.3 配置 …… 201
5.8.4 模块初始化 …… 202
5.8.5 建立连接 …… 204
5.8.6 Broadcast …… 205
5.8.7 Orderer …… 206
5.8.8 Deliver …… 209
5.8.9 orderer 共识机制 …… 210
5.9 channel …… 213
5.9.1 目录 …… 213
5.9.2 配置文件 …… 214
5.9.3 命令 …… 215

第 6 章 区块链政务数据共享及服务 … 220

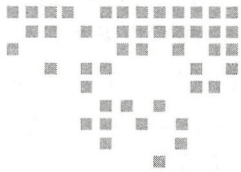
6.1 背景 …… 220
6.2 现有系统面临的挑战 …… 221
6.3 业务需求 …… 221
6.4 系统总体架构设计 …… 222
6.4.1 系统架构设计 …… 222
6.4.2 逻辑架构视图 …… 224
6.4.3 逻辑组网示例 …… 225
6.4.4 物理组网示例 …… 226

6.5 证照办件方案描述 …… 227
6.5.1 场景描述 …… 227
6.5.2 办件消息发布 …… 228
6.5.3 可订阅消息频道查询 …… 229
6.5.4 办件消息订阅 …… 229
6.6 文件共享方案 …… 230
6.6.1 场景描述 …… 230
6.6.2 云存储方案 …… 230
6.6.3 云存储安全保障方案 …… 231
6.7 证照共享方案 …… 232
6.7.1 政务服务数据标准 …… 232
6.7.2 数据上传 …… 235
6.7.3 数据查询 …… 235
6.8 系统接口设计 …… 238
6.8.1 保存政务服务数据 …… 238
6.8.2 批量保存政务服务数据 …… 239
6.8.3 查询政务服务数据 …… 240
6.8.4 发送消息 …… 241
6.8.5 获取附件 …… 242
6.8.6 获取可订阅消息 …… 245
6.9 系统功能设计 …… 246
6.9.1 总体功能结构 …… 246
6.9.2 政务服务数据业务功能 …… 247
6.9.3 平台管理功能 …… 251
6.9.4 系统管理功能 …… 255
6.10 智能合约设计 …… 257
6.10.1 智能合约多层结构设计 …… 257
6.10.2 智能合约模块设计 …… 258
6.10.3 智能合约二次开发 …… 264
6.11 平台的可视化部署 …… 266
6.12 政务数据的三权关系 …… 268

第 7 章 区块链应用设计 …… 270

7.1 区块链在数字商票中的应用 …… 270
7.1.1 简述 …… 270
7.1.2 区块链解决的关键问题 …… 270
7.1.3 方案描述 …… 270
7.1.4 小结 …… 275
7.2 区块链在文化交易中的应用 …… 275
7.2.1 简述 …… 275
7.2.2 区块链解决的关键问题 …… 275
7.2.3 方案描述 …… 276
7.2.4 小结 …… 280
7.3 区块链在烟草溯源中的应用 …… 280
7.3.1 简述 …… 280
7.3.2 区块链解决的关键问题 …… 280
7.3.3 方案描述 …… 281
7.3.4 小结 …… 284
7.4 区块链在海事稽查中的应用 …… 285
7.4.1 简述 …… 285
7.4.2 区块链解决的关键问题 …… 285
7.4.3 方案描述 …… 286
7.4.4 小结 …… 288
7.5 区块链在教育领域的应用 …… 289
7.5.1 简述 …… 289
7.5.2 区块链解决的关键问题 …… 289
7.5.3 方案描述 …… 289
7.5.4 小结 …… 290
7.6 区块链在审计领域的应用 …… 290
7.6.1 背景 …… 290
7.6.2 区块链解决的关键问题 …… 291
7.6.3 方案描述 …… 292

7.6.4 小结 …… 292
7.7 区块链身份认证 …… 293
7.7.1 背景 …… 293
7.7.2 区块链解决的关键问题 …… 295
7.7.3 方案整体架构 …… 296
7.7.4 小结 …… 299
7.8 区块链在数据流通中的应用 …… 299
7.8.1 背景 …… 299
7.8.2 区块链解决的关键问题 …… 300
7.8.3 方案整体架构 …… 302
7.8.4 小结 …… 304
7.9 区块链在供应链金融中的应用 …… 304
7.9.1 背景 …… 304
7.9.2 区块链解决的关键问题 …… 304
7.9.3 方案整体架构（以物流为例）… 305
7.9.4 小结 …… 306
第 8 章 区块链未来展望 …… 307
8.1 区块链与人工智能的关系 …… 307
8.2 区块链与大数据 …… 314
8.3 区块链即服务 …… 316
8.3.1 概念 …… 316
8.3.2 原理 …… 316
8.3.3 IBM 区块链服务 …… 317
8.3.4 微软区块链服务 …… 324
8.3.5 小结 …… 328
8.4 GDPR 对区块链的影响 …… 329
8.5 区块链面临的挑战 …… 332
8.5.1 待解决的四大难题 …… 332
8.5.2 性能问题及解决建议 …… 334
8.5.3 安全问题及解决建议 …… 337
参考文献 …… 340



第 1 章 Chapter 1

区块链基础

区块链技术自问世以来就一直受到全世界的持续关注，有人称之为继蒸汽机、电力和互联网之后的下一代颠覆性核心技术。区块链作为一种新型的技术组合，其分布式、不可篡改、不可抵赖等特点带来了一种全新的信用模式，正在引起各领域对未来应用前景的无限憧憬。

区块链技术是颠覆性的技术革命，是互联网发展到一定程度后的自我进化，其意义远超互联网。应用区块链技术可以将商品或服务的生产者和消费者直接连接到一起，无须中介机构和中间组织的介入，从而减少信息不透明性、提高业务效率、降低成本、减少风险。

区块链对金融、科技、社会等方面都将有重要影响，甚至将改变世界。

1）对科技的改变。区块链的出现，将使得软件、加密、存储、数据、网络等多种传统技术得以创新优化。

2）对流程的改变。区块链能够在保险、银行（DVP、RTGS）、司法等领域改造新流程，从而创造极大的商机。

3）对社会的改变。区块链本质上是分布式去中介互信技术，现在社会上依赖信息不透明而存在的各种中介机构，如房产、人力资源、线上线下电商等，由于区块链技术的应用可能将不再有存在价值。因此，区块链对于社会机构将是一种颠覆性的技术。

区块链起源于一种支持比特币运行的底层技术。

区块链的概念首次在 2008 年年末由中本聪（Satoshi Nakamoto）发表在比特币论坛中的论文“Bitcoin：A Peer-to-Peer Electronic Cash System”提出。论文中的区块链技术是构建比特币数据结构与交易信息加密传输的基础技术，该技术实现了比特币的挖矿与交易。它主要解决了以下 3 个现存的问题：

1）借助第三方机构来处理信息的模式有点与点之间缺乏信任的内生弱点，商家为了提防自己的客户，会向客户索要完全不必要的信息，但仍然不能避免一定的欺诈行为。

2）中介机构的存在增加了交易成本，限制了实际可行的最小交易规模。

3）数字签名本身能够解决电子货币身份问题，如果还需要第三方支持才能防止双重消费，则系统将失去价值。

区块链不是一个单一的、全新的技术，它是多种技术整合的结果。区块链的四大核心技术是：数据结构、分布式存储、加密算法、共识机制。区块链可以理解成一个分布式的数据存储仓库，该仓库是去中心化的。去中心化的一个非常重要的特征是：网络中的所有数据不是由单一的节点存储的，并且网络中发生的交易也不是由该节点进行认证的。

数据机构由“区块”和“链”组成。区块相当于一个数据包，这个数据包中存储了转账交易数据、智能合约代码和执行数据信息等。“链”不是真正意义上的链式结构，是以时间戳为根据，表示网络中数据信息先后次序。

分布式存储表示网络上所有的数据都是公开的，这点区别于传统的中心化思维。网络中的每个节点的基本单位是一个计算机。任何一台在网络上的计算机都可以获得并存储网络上所有的数据。分布式的优势在于所有计算机用于共同的“账本”，单纯的某个节点的数据变更，无法改变其他计算机上的数据，因此改动的数据无法得到网络的认可。只有网络中半数以上的节点同意的数据才会被网络认可为真实的数据。因此如果想要篡改网络中的数据，必须同时控制网络中半数以上的计算机。在网络中写入数据的权利是由工作量证明机制来认证的，这也就是我们所说的挖矿，每个获得记账权的计算机会获得系统奖励的代币，比特币就是其中一种代币。其实，这是一个纳什均衡的过程。因为整个区块链的网络需要全体计算机共同计算维护，维护是需要代价的（比如电力等），因此网络需要对维护系统的机器进行激励。计算机的运算能力在一定程度上可以表示对网络的贡献程度，因此可以用这种方式维持网络的运作。一旦某个计算机获取了记账权利，那么它会立刻向全网广播这条消息，网络中的所有成员都认可该消息，系统会给该计算机一定的代币作为奖励。

加密算法是对交易的对象、数据等进行加密。因为网络中所有的数据都是向全体成员公开的，那么为了证明交易是由某两台计算机进行的，同时为了保证交易的信息不被篡改，需要对数据进行加密处理。加密由一个复杂的、非对称的哈希算法组成，该算法会产生两个密码锁：公钥和私钥。公钥是公开的，用于加密使用；而私钥仅仅用户自己知道，在解密和身份认证时使用。

1.1 区块链常用名词解释

区块链技术（Block Chain）：是指通过去中心化的方式集体维护一个可靠数据库的技术方案。该技术方案主要将区块（Block）通过密码学方法相互关联起来，每个数据块包含了一定时间内的系统全部数据信息，并且生成数字签名以验证信息的有效性，并链接到下一个

数据块形成一条主链（Chain）。

区块（Block）：是区块链中的一条记录，包含并确认待处理的交易。一个区块是一个数据包，其中包含零个或多个交易、前块（“父块”）的散列值，以及可选的其他数据。除了初始的“创世区块”以外，每个区块都包含它父块的散列值。区块的全部集合被称为区块链，并且包含了一个网络里的全部交易历史。注意，有些基于区块链的加密货币使用“总账”这个词语来代替区块链，这二者的意思是大致相同的。在使用“总账”这个术语的系统里，每个区块都通常包括每个账户的目前状态（比如货币余额、部分履行的合约、注册）的全部拷贝，并允许用户抛弃过时的历史数据。

挖矿（Mining）：指通过计算形成新的区块，是交易的支持者利用自身的计算机硬件为网络做数学计算进行交易确认和提高安全性的过程。以比特币为例：交易支持者（矿工）在计算机上运行比特币软件不断地计算软件提供的复杂的密码学问题来保证交易的进行。作为对他们服务的奖励，矿工可以得到他们所确认的交易中包含的手续费，以及新创建的比特币。

对等式网络（Peer-to-Peer Network）：是指通过允许单个节点与其他节点直接交互，从而实现整个系统像有组织的集体一样运作的系统。以比特币为例：网络以这样一种方式构建——每个用户都在传播其他用户的交易，而且重要的是，不需要银行或其他金融机构作为第三方。

哈希（Hash）：也称散列函数，是一类数学函数，可以在有限合理的时间内，将任意长度的消息压缩为固定长度的二进制串，其输出值称为哈希值，也称为散列值。以哈希函数为基础构造的哈希算法，在现代密码学中扮演着重要的角色，常用于实现数据完整性和实体认证，同时也构成多种密码体制和协议的安全保障。哈希散列是密码学里的经典技术，即把任意长度的输入通过哈希算法，变换成固定长度的、由字母和数字组成的输出。

数字签名（Digital Signature）：是一个可以让人证明所有权的数学机制。数字签名算法是一种用户可以用私钥为文档产生一段叫作签名的短字符串数据的处理，以至于任何拥有相应私钥、签名和文档的人可以验证以下两点：

1）该文件是由特定的私钥的拥有者“签名”的；

2）该文档在签名后没有被改变过。请注意，这不同于传统的签名，在传统签名上你可以在签名后涂抹多余的文字，而且这样做可能无法被分辨；而在数字签名后任何对文档的改变会使签名无效。

公钥加密（public key encryption）：一种特殊的加密，具有在同一时间生成两个密钥的处理（通常称为私钥和公钥），使得利用一个密钥对文档进行加密后，可以用另外一个密钥进行解密。一般地，正如其名字所建议的，个人发布他们的公钥，并给自己保留私钥。

私钥（Private Key）：是一个证明你有权从一个特定的钱包消费电子货币的保密数据块，是通过数字签名来实现的。例如，5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh。

分叉：指向同一个父块的两个区块被同时生成的情况，某些部分的矿工看到其中一个区块，而其他的矿工则看到另外一个区块。这导致两种区块链同时增长。通常来说，随着在一个链上的矿工得到幸运并且那条链增长的话，所有的矿工都会转到那条链上。

硬分叉：是指当比特币协议规则发生改变，旧节点拒绝接受由新节点创造的区块的情况。违反规则的区块将被忽视，矿工会按照他们的规则集，在他们最后见证的区块之后创建区块。

软分叉：是当比特币协议规则发生改变时，旧的节点并不会意识到规则是不同的，它们将遵循改变后的规则集，继续接受由新节点创造的区块。矿工们可能会在他们完全没有理解或者验证过的区块上进行工作。

双重花费：是一个故意的分叉，指一个有着大量挖矿能力的用户发送一个交易来购买产品，在收到产品后又做出另外一个交易把相同量的币发给自己的情况。攻击者创造一个区块，这个区块和包含原始交易的区块在同一个层次上，但是包含的并非原始交易而是第二个交易，并且开始在这个分叉上挖矿。如果攻击者有超过 50% 的挖矿能力的话，双重花费最终可以保证在任何区块深度上成功；低于 50% 的话，有部分可能性成功。但是它经常在深度 2 ～ 5 上有唯一显著的可能。因此，大多数加密货币交易所、博彩站点还有金融服务在接受支付之前需要等待 6 个区块被生产出来（也叫“6 次确认”）。

1.2 区块链的发展历程

1. 区块链 1.0

区块链 1.0 仅是去中心化的数字货币 + 支付行为。其特征是：以区块为单位的链状数据块结构；全网共享账本；非对称加密；源代码开源，主要具备的是去中心化的数字货币和支付平台的功能；目标是去中心化。区块链 1.0 是以比特币为代表的数字货币应用，其场景包括支付、流通等货币职能。

现在很多人对“去中心化”存在很大的理解偏差。“去中心化”的英文单词是 Decentralized。但其实翻译过来为分散，而非去中心化。

区块链是一种软件系统，而软件系统的网络架构一般有 3 种模式：单中心、多中心、分布式。单词 Decentralized 只是表明不是单中心模式，可能为多中心或弱中心，也可能是分布式的。

在中国台湾地区，大多将 Decentralized 翻译为“分散式的”而不是“去中心化”。

所以关于去中心化，绝大多数人还是误解颇深。

所谓去中心化，并不是“消灭所有的中心”。在现实里，实际上是这样的：由“原本只有少量的大中心”，慢慢演化成“有大量的更小规模的中心”，也就是分散（Decentralized）的原意。

典型的区块链 1.0 架构为 BTC，LTC，如图 1-1 所示。

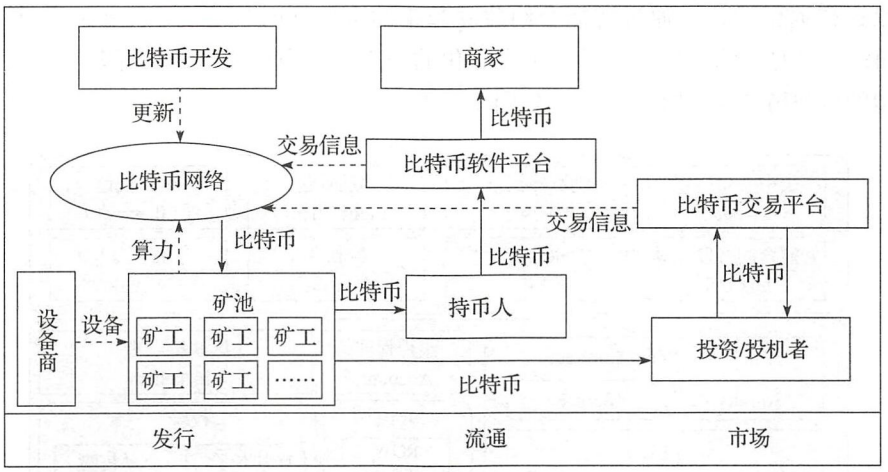


图 1-1　区块链 1.0 架构

区块链 1.0 的局限如下：比特币的 1M 区块大小导致在交易频次越来越高、人们需求越来越多的情况下，转账速度变得越来越慢。这个问题可以由扩容解决，所以出现了之后的比特现金和比特黄金，以及比特钻石等。只满足数字货币的交易和支付功能使得该应用不能被大范围地普及到生活中，给日常生活带来的益处十分有限，区块链的概念也难以深入人心。

2. 区块链 2.0

区块链 2.0 主要的改进之处在于智能合约的开发及应用。

智能合约：区块链系统中的应用，是已编码的可自动运行的业务逻辑，通常有自己的代币和专用开发语言；Dapp，包含用户界面的应用，包括但不限于各种加密货币，如以太钱包；虚拟机，用于执行智能合约编译后的代码，虚拟机是图灵完备的。智能合约已开始在区块链上应用，用机器合约指令代替人工操作，让一切变得更加透明、高效、没有人为操作和干扰。比如以太坊上的艾希欧，就大大降低了融资成本。

区块链 2.0 是数字货币与智能合约相结合，对金融领域更广泛的场景和流程进行优化的应用。其最大的升级之处在于有了智能合约。

智能合约是 20 世纪 90 年代由尼克萨博提出的理念，几乎与互联网同时出现。由于缺少可信的执行环境，智能合约并没有被应用到实际产业中。自比特币诞生后，人们认识到，比特币的底层技术区块链天生可以为智能合约提供可信的执行环境。

以太坊 ETH 首先看到了区块链和智能合约的契合，发布了白皮书《以太坊：下一代智能合约和去中心化应用平台》，并一直致力于将以太坊打造成最佳智能合约平台。所以有人说，比特币引领区块链，以太坊复活智能合约。

所谓智能合约，是指以数字形式定义的一系列承诺，包括合约参与方可以在上面执行这些承诺的协议。智能合约一旦设立指定后，能够无须中介的参与自动执行，并且没有人可以阻止它的运行。

可以这样通俗地说，通过智能合约建立起来的合约同时具备两个功能：一个是现实产生的合同；一个是不需要第三方的、去中心化的公正、超强行动力的执行者。

典型的区块链 2.0 架构为 ETH，如图 1-2 所示。

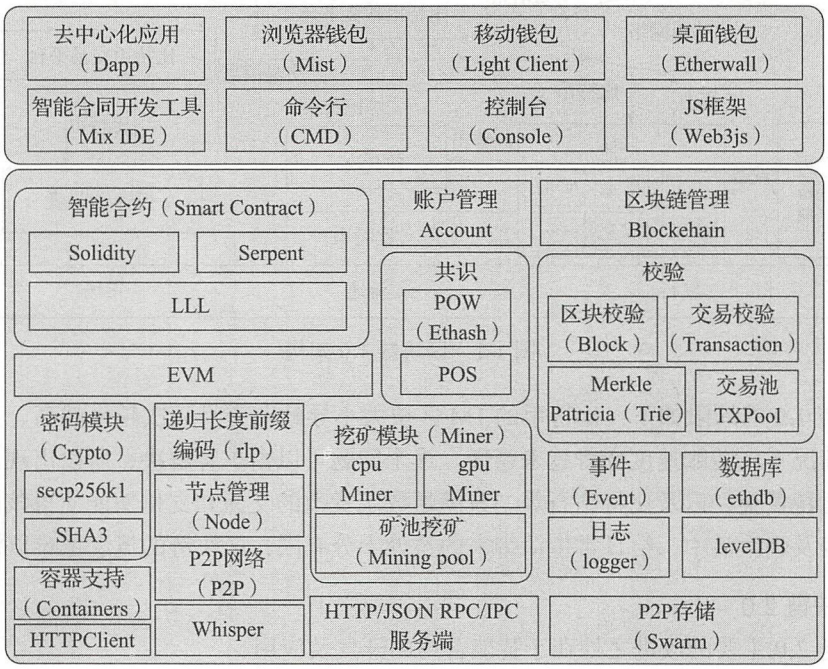


图 1-2　区块链 2.0 架构

3. 区块链 3.0

区块链 3.0 是什么众说纷纭，目前被公认的说法还没有。总之谁可以推动区块链行业的发展，那么谁就是 3.0。区块链 3.0 将是价值互联网的内核，实现跨链通信，极大提升性能，超越智能合约，深入社会的各个领域应用中。区块链会超越金融领域，进入社会公证、智能化领域，主要应用在社会治理领域，包括身份认证、公证、仲裁、审计、域名、物流、医疗、邮件、签证、投票等方面，应用范围扩大到了整个社会，区块链技术有可能成为“万物互联”的一种最底层的协议和操作系统。区块链技术不仅可以成功应用于数字加密货币领域，同时在经济、金融和社会系统中也有广泛的应用场景。根据区块链技术可能的应用场景，可以将区块链的主要应用笼统地归纳为数字货币、数据存储、数据鉴证、金融交易、资产管理和选举投票 6 个场景。

1）数字货币：以比特币为代表，本质上是由分布式网络系统生成的数字货币，其发行过程不依赖特定的中心化机构。

2）数据存储：区块链的高冗余存储、去中心化、高安全性和隐私保护等特点，使其特别适合存储和保护重要隐私数据，以避免因中心化机构遭受攻击或权限管理不当而造成的大

规模数据丢失或泄露。

3）数据鉴证：区块链数据带有时间戳，由共识节点共同验证和记录，不可篡改和伪造，这些特点使得它可广泛应用于各类数据公证和审计场景。例如，区块链可以永久地安全存储由政府机构核发的各类许可证、登记表、执照、证明、认证和记录等。

4）金融交易：区块链技术与金融市场应用有非常高的契合度。区块链可以在去中心化系统中自发地产生信用，能够建立区域布局中心机构信用背书的金融市场，从而在很大程度上实现了“金融脱媒”。同时，利用区块链自动化智能合约和可编程的特点，能够极大地降低成本和提高效率。

5）资产管理：区块链能够实现有形和无形资产的确权、授权和实时监控。无形资产管理方面可广泛应用于知识产权保护、域名管理、积分管理等领域；有形资产管理方面则可结合物联网技术形成“数字智能资产”，实现基于区块链的分布式授权与控制。

6）选举投票：区块链可以低成本高效地实现政治选举、企业股东投票等应用，同时基于投票可广泛应用于博彩、预测市场和社会制造等领域。

未来 3 ～ 5 年，区块链技术将在物联网、金融交易、网络安全、公共记录等多个领域大显身手，显著改进这些领域的服务流程，甚至颠覆这些领域内的传统商业模式，未来发展潜力巨大。区块链技术带来的无处不在的价值交换，使得社会形成一个多种设备的无缝对接的价值互联世界。区块链使得经济不仅仅是金钱的流通，互联网不仅仅是信息的流通，而将进一步促进信息、金钱、价值的有效配置和流通，将人力内耗降到最低，成为真正意义上的去中心化组织。

1.3　区块链概念

1.3.1　区块链是什么

区块链是一种分布式记账技术，比特币是区块链技术的一种数字交易模型系统。基于区块链技术，比特币的所有交易信息并不是储存在某一台服务器或计算机上，区块链技术会将所有交易信息储存在世界上所有加入比特币的计算机上，全量交易信息无处不在，打开任何一台联网的计算机去连接比特币系统，你的交易信息将会在同步后出现。基于区块链技术，比特币上没有设计中心数据库，在一台或几台计算机上修改数据对总交易信息没有影响，也没有中心数据库可以入侵，除非能一举入侵全世界所有安装了比特币的计算机（超千万台）。理论上谁都无法对交易记录造假，也无法抹去交易记录。正是由于具有这个特性，比特币验证了区块链技术会给我们的生活带来更多不一样的可能性。

1.3.2　区块链的特性

结合定义区块链的定义，区块链具有 4 个主要的特性：去中心化、去信任、集体维护、

可靠数据库。并且由这 4 个特征会引申出另外两个特征：开源、隐私保护。如果一个系统不具备这些特征，将不能被视为基于区块链技术的应用。

1. 去中心化（Decentralized）

整个网络没有中心化的硬件或者管理机构，任意节点之间的权利和义务都是均等的，且任一节点的损坏或者失去都不会影响整个系统的运作。因此也可以认为区块链系统具有极好的健壮性。

2. 去信任（Trustless）

参与整个系统中的每个节点之间进行数据交换是无须互相信任的，整个系统的运作规则是公开透明的，所有的数据内容也是公开的，因此在系统指定的规则范围和时间范围内，节点之间是不能也无法欺骗其他节点的。

3. 集体维护（Collectively Maintain）

系统中的数据块由整个系统中所有具有维护功能的节点来共同维护，而这些具有维护功能的节点是任何计算机都可以担任的。

4. 可靠数据库（Reliable Database）

整个系统将通过分数据库的形式，让每个参与节点都能获得一份完整数据库的副本。除非能够同时控制整个系统中超过 51% 的节点，否则在单个节点上对数据库的修改是无效的，也无法影响其他节点上的数据内容。因此参与系统中的节点越多，计算能力越强，该系统中的数据安全性越高。

5. 开源（Open Source）

由于整个系统的运作规则必须是公开透明的，所以对于程序而言，整个系统必定是开源的。

6. 隐私保护（Anonymity）

由于节点和节点之间无须互相信任，因此节点和节点之间无须公开身份，系统中的每个参与的节点的隐私都是受到保护的。

1.3.3 区块链分类

1. 公有链（Public Blockchain）

公有链通常也称为非许可链（Permissionless Blockchain），无官方组织及管理机构，无中心服务器，参与的节点按照系统规格自由接入网络，不受控制，节点间基于共识机制开展工作。

公有链是真正意义上的完全去中心化的区块链，它通过密码学保证交易不可篡改，同时也利用密码学验证以及经济上的激励，在互为陌生的网络环境中建立共识，从而形成去中心化的信用机制。在公有链中的共识机制一般是工作量证明（PoW）或权益证明（PoS），用

户对共识形成的影响力直接取决于他们在网络中拥有资源的占比。

公有链一般适合于虚拟货币、面向大众的电子商务、互联网金融等 B2C、C2C 或 C2B 等应用场景。比特币和以太坊等就是典型的公有链。

2. 联盟链（Consortium Blockchain）

联盟链是一种需要注册许可的区块链，这种区块链也称为许可链（Permissioned Blockchain）。联盟链仅限于联盟成员参与，区块链上的读写权限、参与记账权限按联盟规则来制定。整个网络由成员机构共同维护，网络接入一般通过成员机构的网关节点接入，共识过程由预先选好的节点控制。由于参与共识的节点比较少，联盟链一般不采用工作量证明的挖矿机制，而采用权益证明或 PBFT（Practical Byzantine Fault Tolerant）、RAFT 等共识算法。

一般来说，联盟链适合于机构间的交易、结算或清算等 B2B 场景。例如，在银行间进行支付、结算、清算的系统就可以采用联盟链的形式，将各家银行的网关节点作为记账节点，若网络上有超过 2/3 的节点确认一个区块，该区块记录的交易将得到全网确认。联盟链对交易的确认时间、每秒交易数都与公有链有较大的区别，对安全和性能的要求也比公有链高。

3. 私有链（Private Blockchain）

私有链建立在某个企业内部，系统的运作规则根据企业要求来设定。

私有链的应用场景一般是企业内部的应用，如数据库管理、审计等。在政府部门也会有一些应用，比如政府的预算和执行，或者政府的行业统计数据，这些一般由政府登记，但公众有权力监督。私有链的价值主要是提供安全、可追溯、不可篡改、自动执行的运算平台，可以同时防范来自内部和外部对数据的安全攻击，这在传统的系统中是很难做到的。

1.3.4　区块链构建信任

在当今这个高度中心化的社会中，我们为了证明自己的信用付出了太多的代价。由于互不信任，我们建立了银行来集中管理我们的金钱，建设了支付宝来做网上交易的中介，这一切的中介都是因为我们不相信自己不认识的另一个人，只有通过中介我们才能建立起脆弱的信任，为此我们付出了大量的无谓的代价。

那有什么可以真正建立起人与人之间的信任呢？基于对某打车平台的信任，我可以将自己车上的空位出租出去，让一个陌生人坐上我的车，这是我们发展共享经济的第一步。但是我们可以像一些国家那样用 Airbnb 将我们多余的房间出租给陌生人吗？可能很难。那我们可以再进一步将我们的床位、车库、厨房、乐器、计算机这些私人闲置物品出租出去吗？恐怕更是难上加难，为什么？因为我们不信任他人，我们不知道出租出去后会发生什么事情。

但是，区块链可以做到。试想，若你每天的所作所为都被大数据记录，被存储在区块链的一个个节点上，无法篡改、无法修饰，那么任何人都可以用区块链来描绘你的数字画

像，通过区块链的评分来确定你这个人是否可信。一旦这个体系建立起来，大量的中介将会失去其存在的价值，而信任一个人也变得不再复杂。

1.3.5 区块链的社会价值

由于人与人之间的信任不够，我们的共享经济只能停留在较简单的事情上。但是当一个人的信用开始由区块链确定之后，那么我们的生活也许将由此改变，万物互联，共享生活将会变得简单，我们的生活将会在一些方面出现质的改变。

1）来自互联网的欺诈将会减少。相信大多数人对于近年来的百度互联网事件心有余悸，百度贴吧的买卖事件让人们不再相信贴吧里面所谓的网友；魏某某的死亡事件让百度的搜索结果令人质疑。所有人都在感叹，在这样的互联网时代，还有什么值得相信？但是区块链可以解决，当每个人都在互联网上成为一个被区块链描绘的人时，那么你的所作所为都将被记录，人们对于企业的信任将会转移为对他人的信任，一旦你帮助你的企业欺骗他人，将会留下你自己的信用污点，那么无论是谁都不会选择铤而走险。

2）真正的共享经济将会建立。每时每刻，我们的周围都有着太多的闲置的东西，这些东西可能是我们的日常生活用品，可能是我们自己的某些财产，甚至是我们自己的技能和知识。这些东西可能我们一年只使用一次，使用之后就成为闲置的资源，除了占据空间外别无他用。当区块链构建起人与人之间的信用后，你可以将你的闲置资源出借或者出租，真正让闲置的资源发挥出其应有的价值。

3）共享金融的真正诞生。如果说预防欺诈仅仅是建立信任的第一步的话，那么实现物品或知识的共享也只能说是共享经济的初级形态。而更深一步的则是基于共享经济，基于区块链对个人金融的一次革命。因为区块链的存在，我们可以用设立区块链的形式消耗计算资源给全网作证，你的数据将不可篡改。你既可以用你的区块链数据向世界上任何一家银行申请信用贷款，也可以通过区块链向世界上任何你不认识的人申请信用借款。在这个时候金融将不再是银行的金融，而是草根的金融，即不通过银行，也能实现资金的融通。而这一点就建立在区块链信用的基础之上。

可以说区块链带来了一个全新时代，一个真正信用的时代，未来的信用是靠全网的公正及每个人智能设备的记账来实现的，一旦真正完成了，我们将万物互联，互相信任，共享生活将真正从口号变为现实。区块链的应用如图 1-3 所示。

1.4 区块链核心技术

1.4.1 综述

1. 区块链核心技术一：拜占庭协定

拜占庭帝国拥有巨大的财富，周围 10 个邻邦对其垂涎已久，但由于拜占庭高墙耸立，

固若金汤，没有一个单独的邻邦能够成功入侵。任何单个邻邦的入侵都会失败，同时也有可能自身被其他 9 个邻邦入侵。拜占庭帝国防御能力如此之强，至少要有 10 个邻邦中的一半以上同时进攻，才有可能将其攻破。然而，如果其中的一个或者几个邻邦本身答应好一起进攻，但实际过程出现背叛，那么入侵者可能都会被歼灭。于是每一方都小心行事，不敢轻易相信邻国。这就是拜占庭将军问题。在这个分布式网络里，每个将军都有一份与其他将军实时同步的消息账本，账本里每个将军的签名都是可以验证身份的。如果有哪些消息不一致，马上就可以知道消息不一致的是哪些将军。尽管有消息不一致的，只要有超过半数的将军同意进攻，少数服从多数，即可达成共识。由此，在一个分布式的系统中，尽管有坏人，坏人可以做任意事情（不受 protocol 限制），比如不响应、发送错误信息、对不同节点发送不同决定、不同错误节点联合起来干坏事等，但是，只要大多数人是好人，就完全有可能去中心化地实现共识。

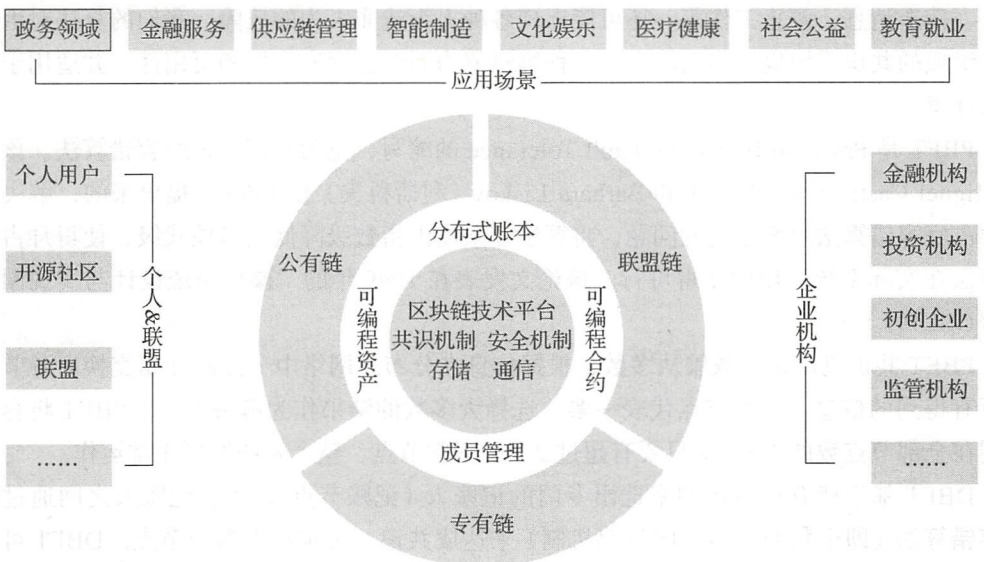


图 1-3　区块链的应用

2. 区块链核心技术二：非对称加密技术

在上述拜占庭将军问题中，如果 10 个将军中的几个同时发起消息，势必会造成系统的混乱，造成各说各的攻击时间或方案，行动难以一致。谁都可以发出进攻的信息，但由谁来发出呢？其实这只要加入一个成本就可以了，即：一段时间内只有一个节点可以传播信息。当某个节点发出统一进攻的信息后，各个节点收到发起者的信息必须签名盖章，确认各自的身份。在如今看来，非对称加密技术完全可以解决这个签名问题。非对称加密算法的加密和解密使用不同的两个密钥，这两个密钥就是我们经常听到的“公钥”和“私钥”。公钥和私钥一般成对出现，如果信息使用公钥加密，那么需要用与该公钥对应的私钥才能解密。

非对称加密技术工作流程如下：

1）乙方生成一对密钥（公钥和私钥）并将公钥向其他方公开。

2）得到该公钥的甲方使用该密钥对机密信息进行加密后再发送给乙方。

3）乙方再用自己保存的另一把专用密钥（私钥）对加密后的信息进行解密。乙方只能用其专用密钥（私钥）解密由对应的公钥加密后的信息。

在传输过程中，即使攻击者截获了传输的密文，并得到了乙方的公钥，也无法破解密文，因为只有乙方的私钥才能解密密文。同样，如果乙方要回复加密信息给甲方，那么需要甲方先公布自己的公钥给乙方用于加密，甲方自己保存的私钥仅用于解密。

3. 区块链核心技术三：容错问题

我们假设，在此网络中消息可能会丢失、损坏、延迟、重复发送，并且接收的顺序与发送的顺序不一致。此外，节点的行为可以是任意的：可以随时加入、退出网络，可以丢弃消息、伪造消息、停止工作等，还可能出现各种人为或非人为的故障。我们的算法对由共识节点组成的共识系统提供容错能力，这种容错能力同时包含安全性和可用性，并适用于任何网络环境。

PBFT 是 Practical Byzantine Fault Tolerance 的缩写，意为实用拜占庭容错算法。该算法是 Miguel Castro（卡斯特罗）和 Barbara Liskov（利斯科夫）在 1999 年提出来的，解决了原始拜占庭容错算法效率不高的问题，将算法复杂度由指数级降低到多项式级，使得拜占庭容错算法在实际系统应用中变得可行。该论文发表在 1999 年的“操作系统设计与实现国际会议”（OSDI99）上。

PBET 共识机制是少数服从多数，根据信息在分布式网络中节点间互相交换后各节点列出所有得到的信息，一个节点代表一票，选择大多数的结果作为解决办法。PBET 将容错量控制在全部节点数的 1/3，即只要有超过 2/3 的正常节点，整个系统便可正常运作。

DBFT 基于持有权益比例来选出专门的记账人（记账节点），然后记账人之间通过拜占庭容错算法（即少数服从多数的投票机制）来达成共识，决定动态参与节点。DBFT 可以容忍任何类型的错误，且专门的多个记账人使得每一个区块都有最终性、不会分叉。

4. 区块链核心技术四：一致性算法

在分布式系统中，存在一致性问题。Paxos 算法是业界使用较广泛的解决一致性问题的算法，它采用一种基于消息传递的模型，解决一个分布式系统如何就某个值（决议）达成一致的问题。一个典型的场景是，在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么它们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令是一致的。一个通用的一致性算法可以应用在许多场景中，是分布式计算中的重要问题。节点通信存在两种模型：共享内存和消息传递。Paxos 算法就是一种基于消息传递模型的一致性算法。

不仅在是分布式系统中，凡是多个过程需要达成某种一致的场合都可以使用 Paxos 算法。一致性算法可以通过共享内存（需要锁）或者消息传递实现，Paxos 算法采用的是后者。Paxos 算法适用的几种情况如下：一台机器中多个进程或线程达成数据一致；分布式文件系统或者分布式数据库中多客户端并发读写数据；分布式存储中多个副本响应读写请求的一致性。

5. 区块链核心技术五：共识机制

区块链共识算法主要是工作量证明（Pow）和权益证明。以比特币举例，其实从技术角度来看可以把 PoW 看作重复使用的 Hashcash，生成工作量证明在概率上来说是一个随机的过程。一个矿工开采新的机密货币，生成区块时，必须得到所有参与者的同意，那个矿工必须得到区块中所有数据的 PoW。与此同时矿工还要时时观察调整这项工作的难度，因为对网络要求是平均每 10 分钟生成一个区块。

区块链的自信任主要体现于分布于区块链中的用户无须信任交易的另一方，也无须信任一个中心化的机构，只需要信任区块链协议下的软件系统即可实现交易。这种自信任的前提是区块链的共识机制，即在一个互不信任的市场中，要想使各节点达成一致的充分必要条件是：每个节点出于对自身利益最大化的考虑，都会自发、诚实地遵守协议中预先设定的规则，判断每一笔记录的真实性，最终将判断为真的记录记入区块链之中。换句话说，如果各节点具有各自独立的利益并互相竞争，则这些节点几乎不可能合谋欺骗用户，而当各节点在网络中拥有公共信誉时，这一点体现得尤为明显。区块链技术正是运用一套基于共识的数学算法，在机器之间建立“信任”网络，从而通过技术背书而非中心化信用机构来进行全新的信用创造。

目前区块链的共识机制可分为四大类：工作量证明机制、权益证明机制、股份授权证明机制和 Pool 验证池[⊖]。

（1）工作量证明机制

工作量证明机制即对于工作量的证明，是生成要加入区块链中的一笔新的交易信息（即新区块）时必须满足的要求。在基于工作量证明机制构建的区块链网络中，节点通过计算随机哈希散列的数值解争夺记账权，求得正确的数值解以生成区块的能力是节点算力的具体表现。工作量证明机制具有完全去中心化的优点，在以工作量证明机制为共识的区块链中，节点可以自由进出。大家所熟知的比特币网络就应用工作量证明机制来生产新的货币。然而，由于工作量证明机制在比特币网络中的应用已经吸引了全球计算机大部分的算力，其他想尝试使用该机制的区块链应用很难获得同样规模的算力来维持自身的安全。同时，基于工作量证明机制的挖矿行为还造成了大量的资源浪费，达成共识所需要的周期也较长，因此该机制并不适合商业应用。

（2）权益证明机制

2012 年，化名 Sunny King 的网友推出了 Peercoin，该加密电子货币采用工作量证明机制发行新币，采用权益证明机制维护网络安全，这是权益证明机制在加密电子货币中的首次

⊖ 唐文剑，吕雯，等 . 区块链将如何重新定义世界［M］. 北京：机械工业出版社，2016.6.

应用。与要求证明人执行一定量的计算工作不同，权益证明要求证明人只提供一定数量加密货币的所有权即可。权益证明机制的运作方式是，当创造一个新区块时，矿工需要创建一个“币权”交易，交易会按照预先设定的比例把一些币发给矿工本身。权益证明机制根据每个节点拥有代币的比例和时间，依据算法等比例地降低节点的挖矿难度，从而加快了寻找随机数的速度。这种共识机制可以缩短达成共识所需的时间，但本质上仍然需要网络中的节点进行挖矿运算。因此，权益证明机制并没有从根本上解决工作量证明机制难以应用于商业领域的问题。

（3）股份授权证明机制

股份授权证明权益证明机制（DPoS）是一种新的保障网络安全的共识机制。它在尝试解决传统的工作量证明机制和权益证明机制问题的同时，还能通过实施科技式的民主抵消中心化所带来的负面效应。

股份授权证明机制与董事会投票类似，该机制拥有一个内置的实时股权人投票系统，就像系统随时都在召开一个永不散场的股东大会，所有股东都在这里投票决定公司决策。基于 DPoS 机制建立的区块链的去中心化依赖于一定数量的代表，而非全体用户。在这样的区块链中，全体节点投票选举出一定数量的节点代表，由它们来代理全体节点确认区块、维持系统有序运行。同时，区块链中的全体节点具有随时罢免和任命代表的权力。如果必要，全体节点可以通过投票让现任节点代表失去代表资格，重新选举新的代表，实现实时的民主。

股份授权证明机制可以大大缩小参与验证和记账节点的数量，从而达到秒级的共识验证。然而，该共识机制仍然不能完美解决区块链在商业中的应用问题，因为该共识机制无法摆脱对于代币的依赖，而在很多商业应用中并不需要代币的存在。

（4）Pool 验证池

Pool 验证池基于传统的分布式一致性技术建立，并辅之以数据验证机制，是目前区块链中广泛使用的一种共识机制。

Pool 验证池不依赖代币就可以工作，在成熟的分布式一致性算法（Pasox、Raft）基础之上，可以实现秒级共识验证，更适合有多方参与的多中心商业模式。不过，Pool 验证池也存在一些不足，例如该共识机制能够实现的分布式程度不如工作量证明机制等。

6. 区块链核心技术六：分布式存储

分布式存储是一种数据存储技术，通过网络使用每台机器上的磁盘空间，并将这些分散的存储资源构成一个虚拟的存储设备，数据分散地存储在网络中的各个角落。所以，分布式存储技术并不是每台计算机都存放完整的数据，而是把数据切割后存放在不同的计算机里。就像有 100 个鸡蛋，不是全放在同一个篮子里，而是分开放在不同的地方，加起来鸡蛋的总和是 100 个。

区块链技术原理的来源可归纳为一个数学问题：拜占庭将军问题。拜占庭将军问题延伸到互联网生活中来，其内涵可概括为：在互联网大背景下，当需要与不熟悉的对手方进行价值

交换活动时，人们如何才能防止不会被其中的恶意破坏者欺骗、迷惑，从而做出错误的决策。进一步将拜占庭将军问题延伸到技术领域中来，其内涵可概括为：在缺少可信任的中央节点和可信任的通道的情况下，分布在网络中的各个节点应如何达成共识。区块链技术解决了闻名已久的拜占庭将军问题——它提供了一种无须信任单个节点还能创建共识网络的方法。

1.4.2　区块链结构

区块链技术并不是一种单一的、全新的技术，而是多种现有技术（如加密算法、P2P 文件传输等）整合的结果。这些技术与数据库巧妙地组合在一起，形成了一种新的数据记录、传递、存储与呈现的方式。简单说，区块链技术就是一种大家共同参与记录信息、存储信息的技术。过去，人们将记录、存储数据的工作交给中心化的机构来完成，而区块链技术则让系统中的每一个人都可以参与数据的记录、存储。区块链技术在没有中央控制点的分布式对等网络下，使用分布式集体运作的方法，构建了一个 P2P 的自组织网络。通过复杂的校验机制，区块链数据库能够保持完整性、连续性和一致性，即使部分参与人作假也无法改变区块链的完整性，更无法篡改区块链中的数据。

区块链技术涉及的关键点包括：去中心化（Decentralized）、去信任（Trustless）、集体维护（Collectively maintain）、可靠数据库（ReliableDatabase）、时间戳（Time stamp）、非对称加密（AsymmetricCryptography）等。

1. 区块与链

区块（Block）：在区块链技术中，数据以电子记录的形式被永久储存下来，存放这些电子记录的文件我们就称之为“区块”（Block）。区块是按时间顺序一个一个先后生成的，每一个区块记录下它在被创建期间发生的所有价值交换活动，所有区块汇总起来形成一个记录合集。

区块结构（BlockStructure）：区块中会记录下区块生成时间段内的交易数据，区块主体实际上就是交易信息的合集。每一种区块链的结构设计可能不完全相同，但大结构上分为块头（header）和块身（body）两部分。块头用于链接到前面的块，并且为区块链数据库提供完整性的保证；块身则包含了经过验证的、块创建过程中发生的、价值交换的所有记录。

区块结构有两个非常重要的特点：首先，每一个区块上记录的交易是上一个区块形成之后、该区块被创建前发生的所有价值交换活动，这个特点保证了数据库的完整性。其次，在绝大多数情况下，一旦新区块完成后被加入区块链的最后，则此区块的数据记录就再也不能被改变或删除。这个特点保证了数据库的严谨性，即无法被篡改。

顾名思义，区块链就是将区块以链的方式组合在一起，以这种方式形成的数据库我们称之为区块链数据库。区块链是系统内所有节点共享的交易数据库，这些节点基于价值交换协议参与到区块链的网络中来。

由于每一个区块的块头都包含了前一个区块的交易信息压缩值，这就使得从创世块（第一个区块）到当前区块连接在一起形成了一条长链。由于如果不知道前一区块的“交易缩影”

值，就没有办法生成当前区块，因此每个区块必定按时间顺序跟随在前一个区块之后。这种所有区块包含前一个区块引用的结构让现存的区块集合形成了一条数据长链，如图 1-4 所示。

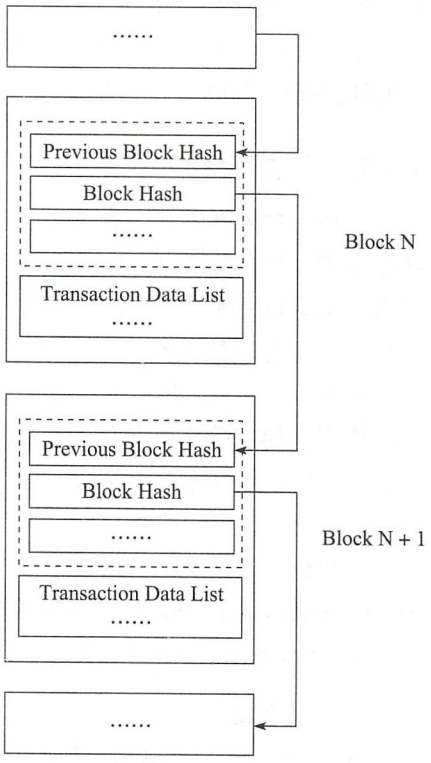


图 1-4　区块链结构

2. 分布式架构

构建一整套协议机制，让全网每一个节点在参与记录的同时也来验证其他节点记录结果的正确性。只有当全网大部分节点（或甚至所有节点）都同时认为这个记录正确时，或者所有参与记录的节点都比对结果一致通过后，记录的真实性才能得到全网认可，记录数据才允许被写入区块中。

构建一个分布式架构的网络系统，让数据库中的所有数据都实时更新并存放于所有参与记录的网络节点中。这样即使部分节点损坏或被黑客攻击，也不会影响整个数据库的数据记录与信息更新。

区块链根据系统确定的开源的、去中心化的协议，构建了一个分布式的结构体系，让价值交换的信息通过分布式传播发送给全网，通过分布式记账确定信息数据内容，盖上时间戳后生成区块数据，再通过分布式传播发送给各个节点，实现分布式存储。医疗区块链的一个分布式架构示意图，如图 1-5 所示。

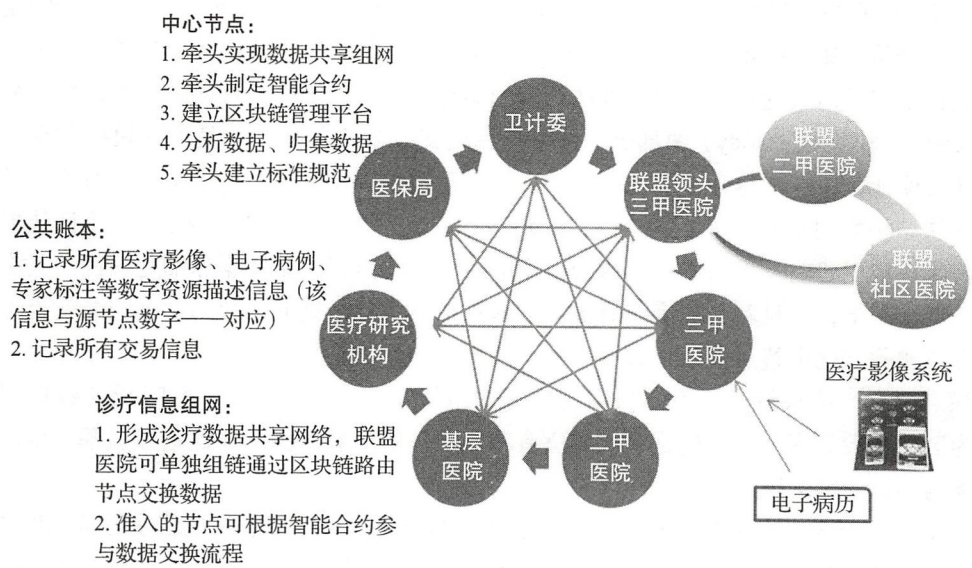


图 1-5　医疗区块链分布式架构

3. 非对称加密算法

加密时的密码被称为“公钥”是公开的全网可见的，所有人都可以用自己的公钥来加密一段信息；解密时的密码被称为“私钥”，是只有信息拥有者才知道的，被加密过的信息只有拥有相应私钥的人才能够解密。

区块链系统内，所有权验证机制的基础是非对称加密算法。常见的非对称加密算法包括 RSA、ElGamal、D-H、ECC（椭圆曲线加密算法）等。

在非对称加密算法中，一个“密钥对”中的两个密钥满足以下两个条件，对信息用其中一个密钥加密后，只有用另一个密钥才能解开；其中一个密钥公开后，根据公开的密钥别人也无法算出另一个。那么我们就称这个密钥对为非对称密钥对，公开的密钥称为公钥，不公开的密钥称为私钥。

在区块链系统的交易中，非对称密钥的基本使用场景有以下两种：

1）公钥对交易信息加密，私钥对交易信息解密。私钥持有人解密后，可以使用收到的价值。

2）私钥对信息签名，公钥验证签名。通过公钥签名验证的信息确认为私钥持有人发出。

1.4.3　智能合约

智能合约是合约的程式化表述，是一段应用程序的代码，可以对外提供服务。从这一点上看，智能合约跟互联网上的其他应用并没有区别。但是从技术层面深入分析，我们认为有以下几点值得特别关注：

1）智能合约是部署到区块链平台上的分布式应用。基于这一点，所有人都可以看到智能合约以及其运行的状态，也就是合约对所有人是透明的。

2）智能合约不依赖某个特定的硬件设备，智能合约的代码由所有参与者的设备来执行。

智能合约目前还是初级阶段，比如比特币中只是转账交易，在业界也有很多人认为比特币的转账交易不能被称为智能合约。以太坊设想把全世界的计算机组成一台世界性的计算机，每条交易要消耗 gas，所设计的智能合约要考虑高昂的运行成本和安全问题，否则会发生 TheDAO 事件。我们认为未来的智能合约需要考虑实现以下目标：

（1）面向专家的智能合约

合约要容易编写，合约的表达性要强，即很容易让看合约的人明白。各个行业内的专家通常都是知识的拥有者，但不一定是一个程序员。这些知识需要通过编写合约才能被应用和传递产生价值。所以合约的编写方法应该是“容易的”“通用的”和“多样化的”，使得任何人都可以进行合约的编写，例如律师、会计、医生等，而不单单只是程序员。

（2）面向业务、面向问题的智能合约

合约的编写或者规则的制定是为了解决问题的，所以在合约的制作过程中不应该因外

界的因素而影响问题的解决。即编写合约的专家全部精力应该用在解决问题上，而无须关心底层的操作需要消耗多少 gas、用什么数据库、两个区块链之间怎么解决跨链等问题。

（3）面向领域的解决方案

合约的模块代表知识的模块，知识模块可以被复用、验证，并且用以构建更大、更复杂的知识。合约编写后应该可以被快速地复制和重用，合约的建立是在合约模块的基础之上，合约的模块可以作为单独的解决方案，也可以被“拼凑”到大的合约之中。软件开发或者程序开发也可以在一定程度上模块化，然而软件的模块化表达性要远远低于合约执行图的表达性，从而导致在可复制性、可重用性等方面的低效。

（4）可科学验证的智能合约

合约不可避免会出现一些软件错误，而程序的逻辑性错误几乎不可能通过软件测试的方法被发现，未来的智能合约需要科学的验证方法来保障其正确性。

（5）合约的法律意义

合约的本意是指参与的多方要共同遵守的规则，当一方违约时应受到法律的制裁，所以现实中的合约是具备法律意义的。但用程序代码表达的智能合约是无法具有法律意义的，从而无法保护参与者，这样的合约难以被广泛地应用到大型的产业应用中。随着技术和社会的进一步发展，我们期待在智能合约的法律意义上取得突破。

（6）可交互的智能合约

作为业务规则的智能合约，也可以代表一个决策的过程。例如一个专家医生可能通过合约运行一个青光眼的手术诊疗决策，合约辅助诊疗过程。合约运行到某一步所推荐的决策是通过理论支撑的，这时候需要把推理的结果和对此结果对应的支撑条件展现给医生，并且同时给出可供选择的其他决策。医生此时是可以经验来覆盖计算机系统的决策的，所以需要可交互式的智能合约。

（7）可信数据交换

数据交换作为数据获取的最重要的一种基本方法是需要通过可信的智能合约来实现，通过合约实现的数据交换需要具备跨链的技术能力，从而实现不同组织之间的数据交换。这里强调支持跨链的智能合约，也就是要求合约不会依赖于某个具体的区块链平台，而应该与区块链平台有一种分离的架构。

（8）可信的业务协同

业务协同技术是联通企业和促进合作的桥梁。企业与企业之间、政府与企业之间、政府部门之间，甚至政府与政府之间、国与国之间都需要可信的业务协同，可信的业务协同是建立在可信的数据交换之上的，是可以通过智能合约来实现的。

基于以上目标，智能合约的核心技术如下：

1）智能合约和区块链分离的系统架构设计，实现合约的跨链操作，如图 1-6 所示。

智能合约是面向问题空间的，编写出来的智能合约模板要具备最大的通用度。有了通用的合约模板，当业务部门之间需要交互和协作的时候需要合约能在多条区块链上执行，这

仅供非商业用途或交流学习使用

就要求计算过程（智能合约）和数据状态（区块链可信数据以及跨链）相分离。在设计合约的跨链协议的时候，通过资源锁、解锁操作实现跨区块链的操作和资源调度。

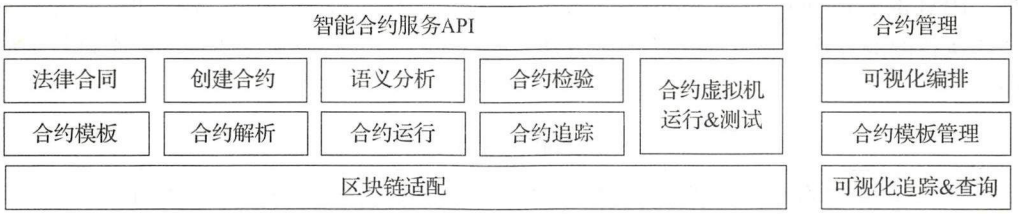


图 1-6　智能合约架构图

2）智能合约的模板化。

在智能合约的开发框架内具备编写模板的能力，可以编写出能重复利用的合约模板，由多个模板可以拼凑出具有更多计算能力和不同计算过程的模板。用户签订合约模板后它就成为一个具体的合约实例，也就是执行中的智能合约。模板是具有通用性的一类合约，比如一个基金清算的模板，一个利率计算的模板。将多套模板实例化后，用户可以生成多个具体的实例合约。

3）可视化的智能合约开发环境。

将智能合约的设计开发过程通用化，并提供可视化的开发环境。业界研究的一个方向是将合约的设计过程通过合约执行图的方法实现，合约执行图中的最小组件是：查询、动作、决策和计划，这 4 个组件在一个用户友好的界面中以拖曳的方式完成，使得编写合约的人无须懂得程序编写，而将全部精力集中在解决业务的思考中，专心构建任务与任务之间的逻辑关系。这就大大简化了对合约的编写要求，提高二次开发的能力，增加合约的通用性。这 4 个基本组件是基于推理的决策系统中所抽象出来的图灵完备组件，类似的语言系统已经在医疗等决策系统中得到验证，并取得业界的认可。

4）提供基于任务流的合约虚拟机，将智能合约的执行过程通用化。

合约的组件或者合约的模块在业务层面也可以理解为任务，合约最终会翻译成合约组件或者任务队列。合约分布式地部署在合约集群机上，这些集群机按照一定的共识算法遍历合约队列和任务队列来执行合约。合约可以在一个或者多个虚拟机中轮番执行，以此保障合约的长效执行方式。

合约执行的过程如图 1-7 所示。

5）提供形式化验证体系，采用科学的方法来验证智能合约的执行过程。

传统的合约验证方法只能是程序测试，但只能抓到有限的程序漏洞，并不能完全保证合约的正确性。这就需要研究合约的科学化的验证方法，来验证合约的正确性，也就是形式化验证方法。合约的科学化验证方法目前业界还处在起步阶段，形式化验证不光是智能合约中难解的问题，也是所有软件的难题。这方面的研究消耗资源、消耗实践，并且常常不能通用，比如操作系统 Windows 就常常出现崩溃的问题。在形式化验证理论中常用的方法，有

仅供非商业用途或交流学习使用

基于图论的，有基于逻辑的定理证明等。其中基于图论的形式化验证方法比较实用，是比较成熟和容易实现的。常用的方法有基于工作流建模的 Petri 图，通过可达性来验证工作流模型；IBM 的 statechart 通过 model checking 来验证工作流模型。实际应用到智能合约，基于合约执行图的方式是借鉴了工作流的方法，是未来进行合约形式化验证的一个可行的研究方向。

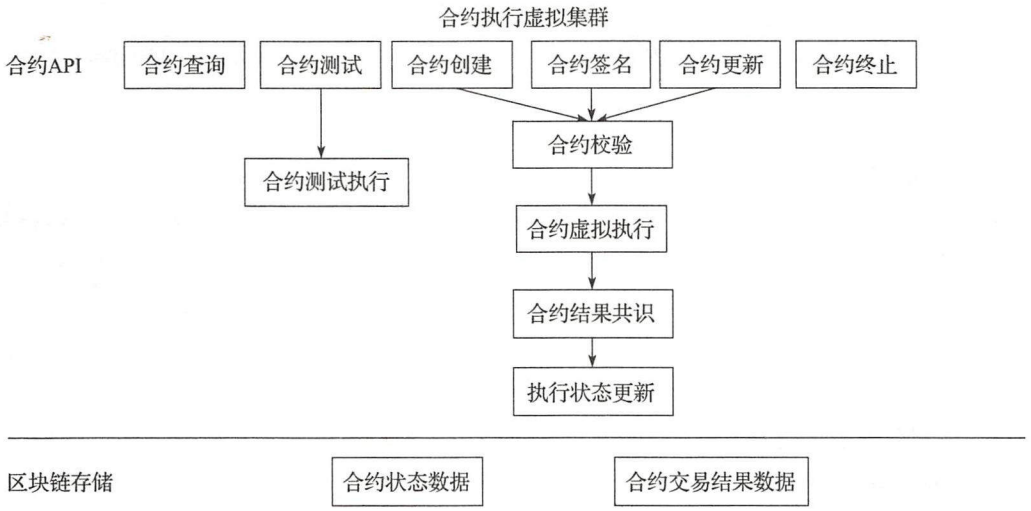


图 1-7 合约执行的过程

1.4.4 跨链技术

区块链属于分布式账本技术的一种，每一条区块链都相当于一个独立的账本，通常情况下不同账本之间是无法实现价值转移的。随着技术以及市场的发展，加密货币的种类越来越多，与此同时也涌现出大量不同的区块链。不同链之间的协同从操作以及价值流通成为用户们的新需求，因此区块链的“跨链技术”应运而生。

所谓“跨链”就是指原本存储在特定区块链上的资产可以转换成为另一条链上的资产，从而实现价值的流通。也可以将其理解为不同资产持有人之间的一种兑换行为，这个过程实际并不改变每条区块链上的价值总额。就好像交易平台提供的币－币交易一样，不同类型的数字货币之间可以进行兑换，只是交易平台的这一行为没有发生在区块链上而已。

从技术上来看区块链属于分布式账本，而从商业层面来看，它本质上属于一种价值网络，不同区块链之间的孤立性不仅导致了数字资产不能在区块链之间流通，同时也将其价值局限在了一个狭隘的范围内，在一定程度上限制了其自身的发展空间。在区块链所面临的诸多问题中，区块链之间互通性极大程度地限制了区块链的应用空间。不论对于公有链还是私有链，跨链技术就是实现价值互联网的关键，它是把区块链从分散的孤岛中拯救出来的“良药”，是区块链向外拓展和连接的桥梁。因此，越来越多的人开始关注跨链技术。2017 年 9

仅供非商业用途或交流学习使用

月，莱特币创始人李启威就曾在推特上表示，莱特币与比特币实现了原子级跨链交换；11月，闪电网络实验室完成了首笔从比特币到莱特币的闪电网络跨链交易。

目前主流的跨链技术包括以下几种。

1. 公证人机制（Notary schemes）

在中心化或多重签名的见证人模式中，见证人是链 A 的合法用户，负责监听链 B 的事件和状态，进而操作链 A。其本质特点是完全不用关注所跨链的结构和共识特性等。假设 A 和 B 是不能进行互相信任的，那么就引入 A 和 B 都能够共同信任的第三方充当公证人作为中介。这样的话，A 和 B 就间接可以互相信任。

公证人机制和现实中的第三方机构相似，充当区块链间共同信任的公证中介的角色。该解决方案中最具代表性的是瑞波实验室于 2012 年提出的 Interledger，这一协议相当于“顶层加密托管系统”，在连接各区块链的同时，实现资金在各分布式账本间的流动。不过从某种意义上来说，具有特权的第三方公证人或许会成为整个系统信任环节中最弱的一环。

瑞波实验室提出的 Interledger 协议旨在连接不同账本并实现它们之间的协同。Interledger 协议适用于所有记账系统，能够包容所有记账系统的差异性，该协议的目标是要打造全球统一的支付标准，创建统一的网络金融传输的协议。

Interledger 协议使两个不同的记账系统可以通过第三方“连接器”或“验证器”互相自由地传输货币。记账系统无须信任“连接器”，因为该协议采用密码算法，用连接器为这两个记账系统创建资金托管，当所有参与方对交易达成共识时，便可相互交易。该协议移除了交易参与者所需的信任，连接器不会丢失或窃取资金，这意味着，这种交易无须得到法律合同的保护和过多的审核，大大降低了门槛。同时，只有参与其中的记账系统才可以跟踪交易，交易的详情可隐藏起来，“验证器”通过加密算法来运行，因此不会直接看到交易的详情。理论上，该协议可以兼容任何在线记账系统，而银行现有的记账系统只需小小的改变就能使用该协议，从而使银行之间无须代理银行就可直接交易。

2. 侧链 / 中继（Sidechains/Relays）

与公证人机制相比，目前跨链技术应用较多的侧链或者中继模式则相对复杂，比如 polkadot 中继链。简单来说，该模式就是以锚定某种原链上的数字货币为基础的新型区块链技术，在保证原有区块链数据隐私性及许可使用的同时，可实现多个区块链间的互相连接。

侧链是用于确认来自于其他区块链的数据的区块链，通过双向挂钩（TwoWay Peg）机制使比特币、Ripple 币等多种资产在不同区块链上以一定的汇率实现转移。

所谓“多种资产在不同区块链上转移”其实并不会实际发生。以比特币为例，侧链的运作机制是，将比特币暂时锁定在比特币区块链上，同时解锁辅助区块链上的等值数字货币；当辅助区块链上的数字货币被锁定时，原先的比特币就被解锁。

侧链进一步扩展了区块链技术的应用范围和创新空间，使区块链支持包括股票、债券、金融衍生品等在内的多种资产类型，以及小微支付、智能合约、安全处理机制、真实世界财

仅供非商业用途或交流学习使用

产注册等，侧链还可以增强区块链的隐私保护。

（1）侧链技术：BTC Relay

侧链是以锚定某种原链上的代币为基础的新型区块链，如同美元锚定到黄金一样。侧链可连接各种链，其他区块链则可以独立存在。但是，现在侧链很难做到在其上建立跨链智能合约，所以很难实现各种金融功能，这正是现有区块链在股票、债券、衍生品等领域尚未取得进展的原因。

BTC Relay 是在以太坊基金会支持之下诞生并成长起来的，它被认为是区块链上的第一个侧链。BTC Relay 把以太坊网络与比特币网络通过以太坊的智能合约连接起来，可以使用户在以太坊上验证比特币交易。它通过以太坊智能合约创建一种小型版本的比特币区块链，但智能合约需要获取比特币网络数据，这还比较难实现去中心化。BTC Relay 进行了跨区块链通信的有意义的尝试，打开了不同区块链交流的通道。

（2）中继技术：Polkadot 和 Cosmos

Polkadot 是由原以太坊主要核心开发者推出的公有链。它旨在解决当今两大阻碍区块链技术传播和接受的难题：即时拓展性和延伸性。Polkadot 计划将私有链 / 联盟链融入公有链的共识网络中，同时又能保有私有链 / 联盟链的原有的数据隐私和许可使用的特性。它可以将多个区块链互相连接。

在 Polkadot 看来，其他区块链都是平行链，Polkadot 则通过中继链（relay-chain）技术能够将原有链上的代币转入类似多重签名控制的原链地址中，对其进行暂时锁定，在中继链上的交易结果将由这些签名人投票决定其是否生效。它还引入了钓鱼人角色对交易进行举报监督。通过 Polkadot 可以将比特币、以太币等都链接到 Polkadot 上，从而实现跨链通信。

Polkadot 目前还是以以太坊为主，实现其与私链的互连，并以其他公有链网络为升级目标，最终让以太坊直接与任何链进行通信。

Cosmos 是 Tendermint 团队推出的一个支持跨链交互的异构网络。Cosmos 采用的 Tendermint 共识算法，类似于实用拜占庭容错共识引擎，具有高性能、一致性等特点。而且在其严格的分叉责任制保证下，能够防止怀有恶意的参与者做出不当操作。

Cosmos 上的第一个空间叫作 Cosmos Hub。Cosmos Hub 中心是一种多资产权益证明加密货币网络，它通过简单的管理机制来实现网络的改动与更新，还可以通过连接其他空间来实现扩展。

如图 1-8 所示是一个非常抽象的系统架构。中间的大圈是 Cosmos Hub，我们把它叫作

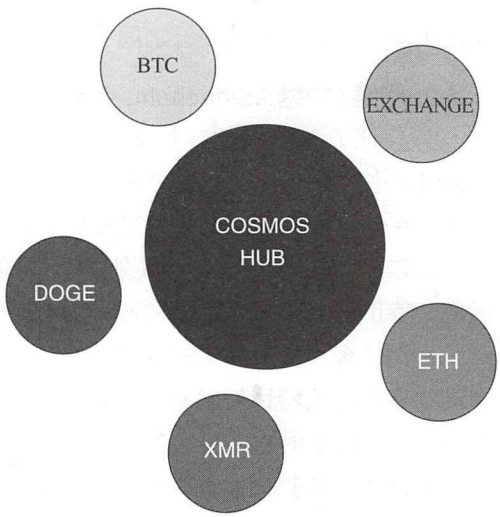


图 1-8　CosmosHub 系统架构图

仅供非商业用途或交流学习使用

Cosmos 枢纽。旁边的这些小圈代表其他的区块链。这些区块链会通过互联链的通信协议与 Cosmos Hub 进行连接。比如比特币、门罗币、以太坊，还有另外一些去中心化的交易所的一些链，理论上都可以连到这个 Hub 上。

Cosmos 网络的中心及各个空间可以通过区块链间通信（IBC）协议进行沟通，这种协议是针对区块链网络的，类似 UDP 或 TCP 网络协议。代币可以安全快速地从一个空间传递到另一个空间，两者之间无须体现汇兑流动性。相反，空间内部所有代币的转移都会通过 Cosmos 中心，它会记录每个空间所持有的代币总量。这个中心会将每个空间与其他故障空间隔离开。因为每个人都可以将新空间连接到 Cosmos 中心，所以 Cosmos 也可以兼容未来新的区块链。

这一架构解决了当今区块链领域面临的许多问题，包括应用程序互操作性、可扩展性，以及无缝更新性。比如，从 Bitcoind、Go-Ethereum、ZCash 或其他区块链系统中衍生出来的空间，都可以接入 Cosmos 中心。这些空间允许 Cosmos 实现无限扩展，从而满足全球交易的需求。下面简单介绍一下这个 IBC 协议，如图 1-9 所示。

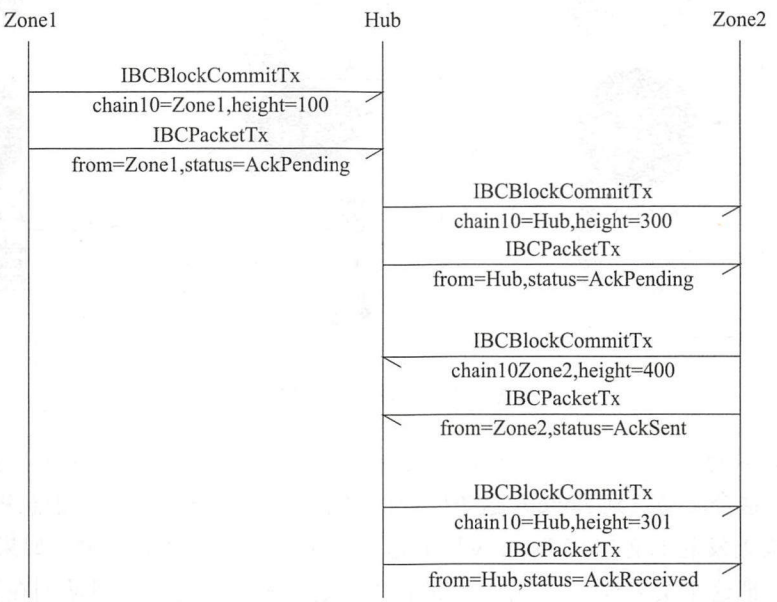


图 1-9　IBC 协议逻辑时序图

IBC 协议定义了最主要的两个交易类型的数据包。一个是 IBCBlockCommitTx，它做的事情实际上就是把发起的这条链的当前最新的区块的头部信息传到目标区块链。这样，目标区块链就获得了当前最新的这个链里面的 Merkle Root。另外一个是 IBCPacketTx。这个传递了跨链转代币的交易信息，这个交易信息是在消息体里面实际包含的 payload 信息，是原链上的一个 Merkle Proof。

以图 1-9 左边为例，如果把 Zone1 当前最新的一个区块的头传递给 Hub，那么 Hub

仅供非商业用途或交流学习使用

就知道它最新的区块的 Merkle Root。当它接着收到一个 IBCPacket 的时候，就可以利用 Packet 中的 Merkle Root 来验证它所包含的 Merkle Proof 是不是正确的。当然隐含的条件就是这个 Hub 是知道 Zone1 当前有效验证者的，也就是说 Hub 知道 Zone1 所有验证人的节点的公钥。它就可以判断头里面的这个信息是有效的，因为它每一个区块头里面都是由超过 2/3 的验证人的私钥签名的。

这个图实际上是一个逻辑时序图。这个消息是怎么从 Zone1 传到 Hub，又从 Hub 传到 Zone2 的呢？是谁发起的这个消息传递呢？这里就涉及一个叫 Relay 的概念，实际上就是在不同的两个区块链之间，它有一个第三方独立的摆渡程序，这个程序负责从原链生成这个 Merkle Proof，组装成 Packet，然后把它摆渡到目标链上。具体的架构图如图 1-10 所示。

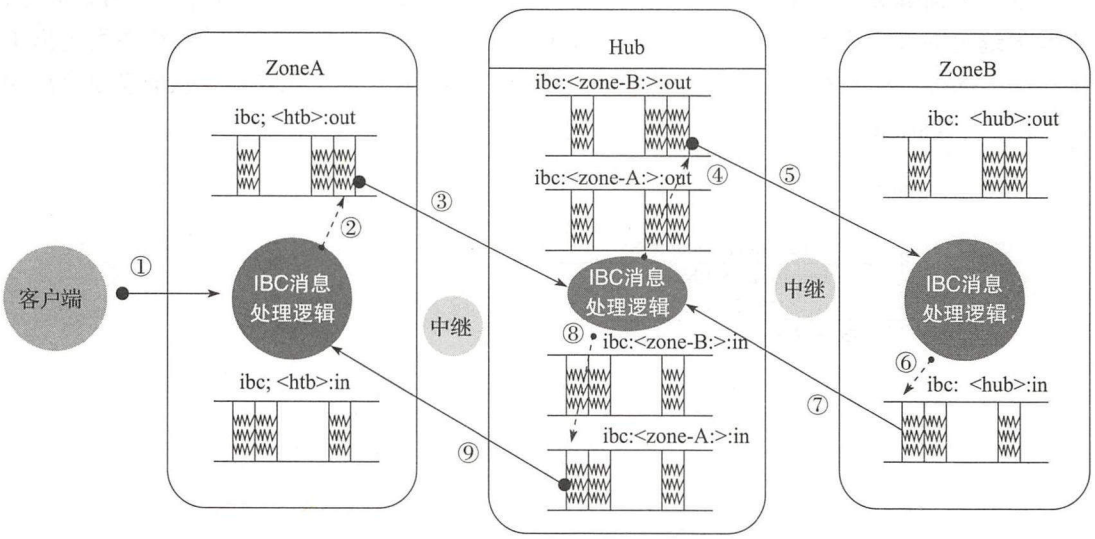


图 1-10 Cosmos 逻辑图

客户端如果想发起一个从 Zone A 到 Zone B 的代币转移。实际上第 1 步就要构造一个这样的交易，这个交易发到 Zone A 这条区块链以后，Zone A 就对它这个消息进行逻辑处理，比方说包括检查发起的客户在 Zone A 里面这种代币有没有足够的数量。如果说是有效的，就到第 2 步，把这个交易放到面向这个 Zone 的消息队列里。这个消息队列在这里显示的是一个先进先出的队列，但实际上它在这个队列里实现是一个 Merkle Tree。

在第 3 步中，中继程序作为 Zone A 的客户端，它实际上一直在监控这个队列。当它看到有新的消息进来的时候，就会生成一个 Merkle Proof。然后把这个作为 IBCPacketTx 的 Payload。第 3 步，把它发到这个 Hub 里面。Hub 对这个消息进行验证，因为 Hub 拥有 Zone A 的当前有效的所有 IBCBlockCommitTx 和所有 Validator 的公钥，即得到所有交易的 Merkle Proof，所以它就可以验证收到的这个 Merkle Proof 是不是有效的，如果是有效的，那就执行第 4 步：给 Zone B 的 Outgoing Queue 里放一个 Message。

仅供非商业用途或交流学习使用

第 5 步和第 3 步非常像，就是在 Hub 和 Zone B 之间也有另外一个中继程序，那这个中继程序也在监控这个 Hub 里的队列，当它发现有新的消息进来的时候，就构造一个这个消息在 Hub 里的 Merkle Proof。然后把这个消息传递到 Zone B 里面。接下来，处理的结果会以收据的方式，如图 1-10 中 6、7、8、9 这样的一个过程。这个高度简化的流程，基本上准确地反映了 IBC 的工作原理。

3. 哈希锁定（Hash-locking）

哈希锁定模式中的哈希锁定协议源于比特币的闪电网络技术，其能起到实时快速小额支付的作用；后来协议内的关键技术——哈希时间锁合约被应用于跨链技术上。总的来说，公证人模式和中继器 / 侧链模式均能支持跨链资产的交换及转移、跨链合约和资产抵押；而哈希锁模式由于不易实现跨链合约，所以其应用场景较为受限，支持功能也相对较少。

Lightning network（闪电网络）提供了一个可扩展的比特币微支付通道网络，它极大提升了比特币网络链外的交易处理能力。交易双方若在区块链上预先设有支付通道，就可以多次、高频、双向地实现快速确认的微支付；双方若无直接的点对点支付通道，只要网络中存在一条连通双方的、由多个支付通道构成的支付路径，闪电网络也可以利用这条支付路径实现资金在双方之间的可靠转移。闪电网络的关键技术是 HTLC 哈希锁定技术，基本原理如下：Alice 和 Bob 达成这样一个协议，协议将锁定 Alice 的 0.1 BTC，在时刻 T 到来之前（T 以未来的某个区块链高度表述），如果 Bob 能够向 Alice 出示一个适当的 R（称为秘密），使得 R 的哈希值等于事先约定的值 H（R），Bob 就能获得这 0.1 BTC；如果直到时刻 T 过去 Bob 仍然未能提供一个正确的 R，这 0.1 BTC 将自动解冻并归还 Alice。

闪电网络并不试图解决单次支付的问题，其假设是单次支付的金额足够小，即使一方违约，另一方的损失也非常小，风险可以承受。因此使用时必须注意“微支付”这个前提。

4. 分布式私钥控制（Distributed private key control）

WanChain 万维链也支持主流公有链间的跨链交易，但首先需要完成在万维链上的注册，确保万维链能够对该链进行唯一识别。对于跨链交易，万维链利用多方计算和门限密钥共享方案。当一种未注册资产由原有链转移到万维链上时，万维链节点会使用一个基于协议的内置资产模板，根据跨链交易信息部署新的智能合约创建新的资产。当一种已注册资产由原有链转移到万维链上时，万维链节点会为用户在已有合约中发放相应等值代币，确保了原有链资产在万维链上仍然可以相互交易流通。

万维链通过分布式的方式完成不同区块链账本的连接及价值交换。它采用通用的跨链协议以及记录跨链交易、链内交易的分布式账本，无论公有链、私有链还是联盟链，均能接入万维链，实现不同区块链账本的连接及资产的跨账本转移。

但是，实现各种链映射到一条链上只是完成了第一步，如果上面的智能合约还是像现在的仅为交易触发，智能合约没办法实现分布式运算和多触发机制，那么多币种智能合约只能实现相当有限的功能。

仅供非商业用途或交流学习使用

早期跨链技术以瑞波和 BTC Relay 为代表，它们更多关注的是资产转移；现有跨链技术以 Polkadot 和 Cosmos 为代表，更多关注的是跨链基础设施。而新出现的 FUSION 实现了多币种智能合约，是一种很有应用价值的公有链，在其上可以产生丰富的跨链金融应用。

1.4.5 ILP 详解及应用

1. 背景

自比特币诞生以来，加密数字货币的区块链网络越来越多，形成了蓬勃发展之势。但是若在不同的区块链之间进行价值转移和交换，就会碰到各种各样的问题。比如，Alice 有“比特币”，想通过“比特币”购买 Bob 的一个笔记本电脑；而笔记本电脑以“瑞波币”来定价，不接受“比特币”。这时 Alice 就得想办法通过一定的兑换，将自己手中的“比特币”换成“瑞波币”，再向 Bob 进行支付。这笔交易的过程应该是：首先 Alice 把“比特币”卖出得到 USD，然后再用 USD 买入“瑞波币”。在这个过程中会有一个币价稳定性问题，币价不稳定将会导致价值损耗。同时交易过程也很烦琐，周期过长。正是针对这样的问题，Ripple 提出了一种跨链价值传输的技术协议 InterLedger Protocal（ILP）。

2. 简介

该技术是 Ripple 实验室于 2012 年提出的，相当于“顶层加密托管系统”，属于跨链技术中的公证人机制的一个实现。ILP 创建了这样一个系统，在这个系统中，两个不同的账本系统可以通过第三方“连接器”来互相自由地转换货币。账本系统无须去信任“连接器”，因为该协议采用密码算法为这两个账本系统和连接器创建资金托管，当所有参与方对资金达成共识时，便可相互交易。ILP 移除了交易参与者所需的信任，连接器不会丢失或窃取资金，这意味着，这种交易无须得到法律合同的保护和过多的审核，大大降低了门槛。

ILP 协议的核心思想在于：“账本”提供的第三方，会向发送者保证，他们的资金只有在“账本”收到证明，且接收方已经收到支付时，才会转给连接者；第三方也同时也向连接者保证，一旦他们完成了协议的最后部分，他们就会收到发送方的资金。

区块链从技术上是去中心化数据库和分布式账本技术，从商业层面则是价值网络。在这个价值网络中，连接的有效节点越多、越分布，可能产生的价值叠加会越大。区块链是价值网络空间的核心基础设施，为了让这种基础设施得到互联互通，可以自由地进行传值转换，需要通过跨链技术，对不同区块链进行连接和扩展，构建价值网络的“高速公路”。

3. 账本

各“账本”要想利用 ILP 技术与其他区块链或者“账本”进行价值转换，必须满足一定的基础条件，才能更好地使自己发行的代币与其他电子货币进行交换，从而被大众接受。

1）必须支持自己“账本”所管理下的一个账户向自己“账本”所管理下另外一个账户进行转账操作；

2）必须支持“托管”式的交易操作，该类型的交易需要两个必要参数：一个执行“托

仅供非商业用途或交流学习使用

管”交易的“原像条件”，一个超时时间；

3）任何用户，不局限于“托管”交易的创建者，在提供“托管”交易的“原像条件”时，就可以决定“托管”交易是被执行还是被拒绝；

4）如果“托管”交易创建后超时，“托管”交易就会自动失效，“托管”交易里所托管的货币会重新进入“托管”交易的创建者账户；

5）“账本”所支持的交易需要具有携带简短消息的功能；

6）“账本”支持事件通知功能，使得各方能及时知道发生了一笔针对自己的“托管”交易。

4. 流程

（1）整个交易流程分为“发送方—> 接收方”与“接收方—> 发送方”两个主流程。每个流程由发生在各个“账本”上的各个子交易（“托管创建”与“托管确认”）组成。“发送方—> 接收方”流程全部是“托管”的创建动作，“接收方—> 发送方”则全由“托管”的确认动作组成。

我们可以了解到，在接收方对“账本 N”上的“托管”交易进行确认前，所有“账本”上被创建的“托管”交易并未被确认，这时每一个“账本”上“托管”交易里指定的源账户并未真正将自己的资产转移出去，而是由“账本”系统进行了托管。只有当接收方进行确认后，在“接收方—> 发送方”的返回流程中，各个“托管”交易才被确认，此时各个“账本”的“托管”交易里指定的目标账户才真正得到源账户的资产，真正发生资产价值转移。

在交易的整个链条过程中，每一个环节上的交易在被确认之前都处于“托管”状态，并没有发生价值转移。即使是连接者去破坏交易，比如说将交易发往另外的地址，发送者也不会有损失，因为最后托管的交易不会得到执行，超时时间条件达成后，交易中的资产会返回给源账户。

当接收者收到一个涉及自己账户的“托管”交易的通知时，接收者会确认交易细节从而确定交易是否正确，然后提供“托管”交易所指定条件的原像，向“账本”系统发出“托管确认”操作，接收者真正得到所需要的加密电子货币。接下来，整个传输链条上的每一个连接者都用同样的“条件原像”去“托管确认”属于自己的“托管”交易，最终每个连接者都收到自己应得的资产价值。

一个值得注意的细节就是：各个“连接者”在创建自己发出的“托管”交易时，设置的超时时间一定要小于自己收到的“托管”交易里所设置的超时时间，从而自己在得到条件的原像时，有时间去执行自己收到的“托管”交易的确认操作，否则会出现自己创建的“托管”交易被接收方或者其他的连接者确认，但自己来不及去确认“属于”自己的“托管”交易，从而造成损失。换句话说，在“账本 B”上把资产转移给了另外一个账户，而在“账本 A”上未确认对应的资产，因此造成资产损失。

（2）我们使用前面提出的场景来描述整个交易的详细步骤，如图 1-11 所示。

对象：发送方—Alice，接收方—Bob，连接者—Cot。

仅供非商业用途或交流学习使用

账本关系：Alice 拥有 bitcoin 的账户，Bob 拥有 ripple 的账户，Cot 拥有 bitcoin 与 ripple 账户。

场景：Alice 要从网上购买 Bob 的笔记本电脑，定价为 29230 个 ripple 币。

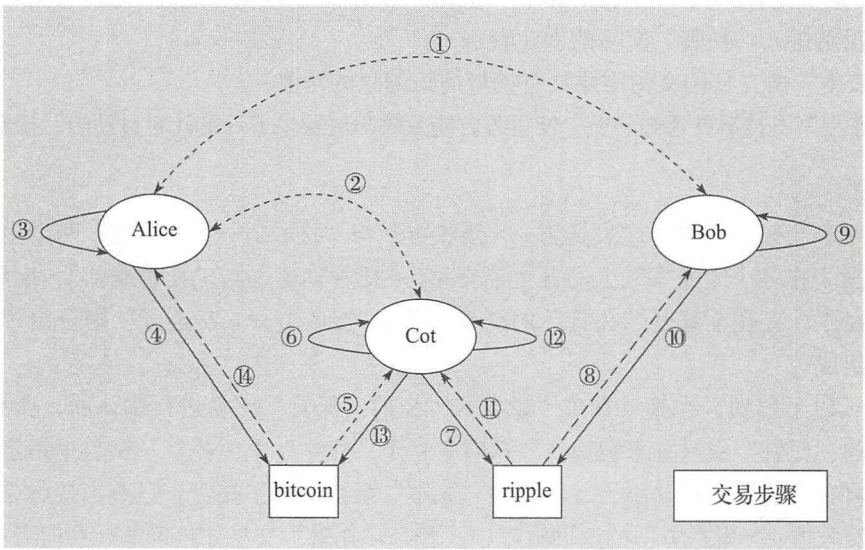


图 1-11　ILP 交易步骤示意图

1）Alice 通过即时通讯软件或者其他通讯手段，得到 Bob 提供的一个“共享密码”。通讯一定要以加密方式进行，使得在沟通后，只有 Alice 与 Bob 知道这个“共享密码”；同时 Bob 会告诉 Alice 自己在 ILP 网络中对应的唯一地址，例如 g.ripple.rHCvhtqhXuBvWt5g79gyXfpG8VcrvUm9E9。

2）Alice 向 Cot 询价，查询自己想发送 29230 个 ripple 币需要多少个 BTC，此时 Cot 会按实时的 BTC 与 ripple 行情算出需要 1 个 BTC，同时 Cot 会多收 0.00001 个 BTC 作为手续费。最终 Alice 得到的询价结果为：需要向 Cot 支付 1.00001 个 BTC。

3）Alice 按 ILP 规定的消息格式生成所需要的 ILP 包，ILP 包里指明目标地址为 Cot。同时基于 ILP 包的私有内容与“共享密码”生成一个“条件原像”，对“条件原像”进行哈希散列，得到一个“托管”交易的“条件”。

4）Alice 在 bitcoin 账本系统上发起一个“托管”创建操作，设置了步骤 3 中的“托管”条件及一个超时时间，同时设置 ILP 包。

5）Cot 在 bitcoin 上监测到一个涉及自己的“托管”创建操作。

6）Cot 解析 ILP 包，计算出自己应该向 Bob 转 29230 个 ripple 币，同时修改 ILP 中的目标地址为 Bob。

7）Cot 在 ripple 账本系统上发起一个“托管”创建操作，设置了步骤 3 中的“托管”条件及一个超时时间（此超时时间要小于步骤 4 中的超时时间）。同时设置 ILP 包。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

8）Bob 在 ripple 上监测到一个涉及自己的“托管”创建操作。

9）Bob 解析 ILP 包，用自己的“共享密码”及 ILP 包里的私有内容生成一个“条件原像”及对应的“条件”。Bob 对比“托管”创建交易里携带的“条件”与自己生成的是否相同，并核实“托管”交易中指定的资产数量是否是 29230，他确认“托管”交易：接收或拒绝。我们这里假定接收。

10）Bob 在 ripple 账本系统上发起一个“托管”确认操作，设置上“条件原像”，ripple 账本上的“托管”交易完成，Bob 收到 29230 个 ripple 币。

11）Cot 在 ripple 上监测到一个涉及自己的“托管”确认操作。

12）Cot 分析“托管”确认操作的内容，得到“条件原像”。

13）Cot 在 bitcoin 账本系统上发起一个“托管”确认操作，设置上“条件原像”，bitcoin 账本上的“托管”交易完成，Cot 收到 1.00001 个 BTC。

14）Cot 在 bitcoin 上监测到一个涉及自己的“托管”确认操作。

从交易流程中我们可以看到，ILP 是利用各个“账本”所提供的“托管”功能实现各子交易的。

如图 1-12 所示，“托管”交易包含 4 个主要步骤。

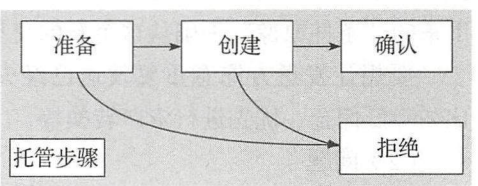


图 1-12　ILP 托管步骤示意图

1）准备：此时什么事情都没发生，只进行了必要数据的准备。

2）创建：隶属于一个“账本”系统上的某个账户的资产被“托管”。

3）确认：“托管”交易完成，资产发生转移，从“账本”系统内的一个账户转移到了另外一个账户。

4）拒绝：“托管”交易被取消，资产回到“账本”系统的源账户。

“托管”功能主要有以下几个特点：

1）“托管”交易被创建后，发送方的资产并未真正转移。

2）“托管”交易不能被撤销或者在一定时间内不能撤销。

3）任何人只要知道“托管”交易列出的“条件原像”，都可以使得“托管”交易被确认，不必要发送方去执行“托管”交易的最终确认操作。

4）任何人只要知道“托管”交易列出的“条件原像”，都可以拒绝“托管”交易，不必要发送方去执行“托管”交易的拒绝操作。

5）“托管”交易可以设置超时时间，若在规定时间内无人进行“确认”操作或者“拒绝”操作，则“托管”交易自动失效。

“托管”交易是 ILP 实现的前提条件，实现“托管”的方式多种多样，可以是“账本”本身具有的功能，也可以是通过双方信任的第三方来代替实现。只有通过“托管”的方式，才能保证各方的利益不受侵害。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

5. 安全

ILP 用了一种带“条件”的交易来保证资产价值在各个“账本”系统之间传递时的安全，发送方用一种加密证明来保证发出去的资产要么被接收方接收，要么返回到发送方的账户。在整个过程中，连接者可能会遭受一定的损失，但是此风险基本可控，风险只与连接者选用的“账本”系统及直接连接的节点有关。

（1）连接者

在 ILP 协议中，连接者是一个重要的角色，支撑起整个跨链网络，使得跨链的资产交换得以正常进行。连接者之所以有积极性是因为在提供资产转换的过程中会收取一定的费用，像做市商时收取的佣金一样有利可图。

但在这个过程中连接者是有一定的风险的，当连接者发出去的“托管”交易被成功确认，但是自己没有及时去确认自己收到的那个“托管”交易，这样本该发给自己的资产就会返回到源账户的地址，从而造成一定的损失。

当连接者未能及时确认发往自己的“托管”交易时，只能靠发送方再次发送带有同样执行“条件”的同样的“托管”交易过来。当连接者收到这个“托管”交易时，再用上次记下来的“条件原像”去确认这个新的交易。

要想让发送方愿意重复发送已经失效的交易，就需要给这样做的发送方一定的奖励，比如降低佣金、优先进行资产转换等。

（2）问题

1）接收方是否会在没有确认涉及自己的“托管”交易时，泄露“条件原像”？

答：如果接收方正常的话，不会有此现象发生。因为这样自己的权益不会被保证，会造成发送方已经发送，接收方收不到资产的情况。这种情况下接收方要承担责任。

2）接收方是否会只确认对方的“托管”交易，却不公布“条件原像”？

答：不会，“托管”确认交易的必要参数之一就是“条件原像”，只要“托管”交易被确认，“账本”系统上就可以查看此交易，从而得到“条件原像”。

3）连接者在未来得及确认涉及自己的“托管”交易，同时发送方又不再次发送，怎么办？

答：没办法，确实无法回避此问题。

4）在发送方与接收方之间有多个连接者的情况下，如果第 1 个连接者先于第 2 个连接者收到了“条件原像”，是否会给第 2 个连接者造成损失？

答：不会，因为第 1 个连接者创建的“托管”交易在一定时间内不能取消，只要在超时时间内第 2 个连接者收到了“条件原像”，就可以确认涉及自己的“托管”交易。

5）谁可以做连接者？

任何拥有两个以上“账本”系统账号的个人都可以做连接者。

6. 核心

大家已经发现了，ILP 并未创建一个统一的“账本”系统，也未要求参与的各方去信任

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

任何个人或者机构，而是依靠现有的、已经存在的“托管”交易来保证各方的利益。那么 ILP 事实上做了什么事呢？

ILP 提出了一种跨链资产转移的方法，但是其首先是一个协议，协议的核心在于以下两点：

1）确定了各个“账本”系统上的每个账户的地址规则，即通过层级关系来确定某一个账号是属于某个范围内某个“账本”上的。

例如 g.ripple.bob，表示公链 ripple 上的一个账户 bob。

2）定义了在跨链交易时的消息传递格式，使得发送方、接收方、连接者之间可以用同样的消息格式进行信息的传递，用以确认收到消息及得到自己的目标地址。

一来使得交易流程中的各个子交易能确定转移的账户，二来使得各方采用统一的格式进行消息传递。

7. 应用场景

1）在只拥有加密电子货币 A 的情况下，以加密电子货币 B 的形式进行支付；

2）拥有不同加密电子货币的账户，想在隶属于不同“账本”的账户之间进行货币的价值转换；

3）同一集团下不同私有链之间进行同类型货币的价值互换。

1.5　热门区块链平台对比分析

1.5.1　分析背景

区块链技术的出现，为智能合约的运行提供了不可抵赖的可信执行环境。以太坊将区块链和智能合约结合，创建了在区块链上实现智能合约、开源的底层系统，极大地便利了区块链应用的开发，并且拓展了区块链技术的应用场景，这就是所谓开启的区块链 2.0 时代。从 2015 年以太坊受到关注至今，大量区块链项目基于以太坊开发，微软等科技巨头也发起成立了以太坊企业联盟（EEA）。

然而，在面向企业级商用的过程中，以太坊共识机制的性能问题、开放网络的隐私性问题等，都在实际运用中面临诸多考验。基于将智能合约与区块链结合的基本理念和技术，众多团体和企业推出了自己的区块链应用平台。本节将选取部分平台，分析其在共识机制、性能、隐私保护、智能合约、技术路线等方面的异同。

1.5.2　平台简介

1. 以太坊

以太坊（Ethereum）是由 Vitalik Buterin 和 Gavin Wood 领导开发的、支持智能合约的去中心化应用平台。以太坊提供图灵完备的脚本语言，极大拓展了区块链技术的应用。项目于

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

2013 年年末发布白皮书启动，2015 年 7 月产生创世区块。

2. EOS

EOS 是由 BM（Daniel Larimer）领导开发的区块链应用平台，其全称是 Enterprise Operation System，是为商用分布式应用设计的一款区块链操作系统。EOS 是引入的一种新的区块链架构，旨在实现分布式应用的性能扩展。首先，EOS 有点类似于微软的 Windows 平台，通过创建一个对开发者友好的区块链底层平台，支持多个应用同时运行，为开发 dAPP 提供底层的模板。其次，EOS 通过并行链和 DPOS 的方式解决了延迟和数据吞吐量的难题，EOS 的处理量是每秒百万级别。最后，EOS 是没有手续费的，普通受众群体更广泛。EOS 上开发 dAPP，需要用到的网络和计算资源是按照开发者拥有的 EOS 的比例分配的。若你拥有了 EOS，就相当于拥有了计算机资源，随着 dAPP 的开发，你可以将手里的 EOS 租赁给别人使用，单从这一点来说 EOS 也具有广泛的价值。简单来说，就是你拥有了 EOS，就相当于拥有了一套房，可以租给别人收取房租，或者说拥有了一块地，可以租给别人建房获取利益。

3. BCOS

BCOS 是微众银行、万向区块链、矩阵元联合创建的企业级应用服务的区块链技术平台，为分布式商业提供完备的区块链技术基础设施及服务。2017 年 7 月，BCOS 第一阶段正式开源。

4. CITA

CITA 是由 EEA（企业以太坊联盟）创始成员之一——Cryptape 秘猿科技自主研发的企业级区块链产品原型。CITA 以高可靠性、高性能、高扩展性以及未来适应性为设计目标，于 2017 年 7 月发布开源版本。

5. Hyperledger

超级账本（Hyperledger）项目是首个面向企业应用场景的开源分布式账本平台。主要包括如下顶级项目。

1）Fabric：包括 Fabric、Fabric CA、Fabric SDK（包括 Node.Js、Python 和 Java 等语言）和 fabric-api 等，目标是区块链的基础核心平台，支持 PBFT 等新的共识机制，支持权限管理。最早由 IBM 和 DAH 发起。

2）Sawtooth：包括 arcade、core、dev-tools、validator、mktplace 等，是 Intel 主要发起和贡献的区块链平台，支持全新的基于硬件芯片的共识机制 Proof of Elapsed Time（PoET）。

3）Iroha：账本平台项目，基于 C++ 实现，具有不少面向 Web 和 Mobile 的特性。主要由 Soramitsu 发起和贡献。

4）Blockchain Explorer：提供 Web 操作界面，通过界面快速查看查询绑定区块链的状态（区块个数、交易历史）信息等。由 DTCC、IBM、Intel 等开发并支持。

仅供非商业用途或交流学习使用

5）Cello：提供区块链平台的部署和运行时管理功能。使用 Cello，管理员可以轻松部署和管理多条区块链；应用开发者可以无须关心如何搭建和维护区块链。由 IBM 团队发起。

6）Indy：提供基于分布式账本技术的数字身份管理机制。由 Sovrin 基金会发起。

7）Composer：提供面向链码开发的高级语言支持，自动生成链码等。由 IBM 团队发起并维护。

8）Burrow：提供以太坊虚拟机的支持，实现支持高效交易的带权限的区块链平台。由 Monax 公司发起支持。

9）Quilt：是 Interledger Protocol（ILP）协议的 Java 实现，是日本的 NTT Data 贡献的。Interledger Protocol 定义了分布式账本与分布式账本之间、传统账本与分布式账本之间的交互过程。

6. MOAC

MOACBlockchain（简称 MoacChain，也被称为 MOAC 区块链、众链之母、女娲链）是一个基于加密协议的开放源码软件，实现在 P2P 网络上支持多种子区块链的公共区块链系统。此区块链中内置的、可挖矿产生的、用于交易的原生加密数字币被称为 MOAC，中文叫墨客。MoacChain 于 2017 年 11 月 22 日正式发布。

1.5.3　类别对比

1. 企业级平台投身联盟链

公有链这类开放的网络环境需要面临复杂的拜占庭将军问题、女巫攻击问题等，因此设计共识算法将以效率换取安全有序，不能满足企业级的高效率应用。而联盟链的形式可以较好地满足企业级应用的需求。

（1）准入机制

在联盟链中，只有获得特定许可的节点才能接入网络。准入机制杜绝了女巫攻击问题。

（2）可信记账节点

只有特定的节点才能从事记账、参与共识算法，这使得传统共识算法有了用武之地，为交易处理的低延迟和吞吐量大幅度增长提供了可能。

2. 非企业平台坚守开放性

以太坊和 EOS 作为个人创始人及其技术团队的创作，坚持“去中心化”的开放理念，采用公有链的形式。全世界任何节点都可读取、发送交易，且交易能获得有效确认，任何节点都能参与其中的共识过程。公有链在一定程度上性能较为低下，但对于区块链的发展仍有积极意义。

（1）规则可信

规则公开、透明、可预见，保护用户免受开发者的影响。在公有链中，程序开发者无权干涉用户，所以区块链可以保护使用其程序的用户。

仅供非商业用途或交流学习使用

（2）访问门槛低

无须类似联盟链搭建应用需要运营方许可，任何拥有足够技术能力的人都可以访问，利于区块链应用的普及。

1.5.4 共识机制对比

1. 从 POW 机制转型成共识

比特币作为首个且具有开放性的比特币网络，采用 PoW（工作量证明）机制，需要网络节点运用算力解决复杂数学谜题的方式分配记账权，保证了比特币网络在没有中心化机构治理的情况下安全地运行。

然而开放网络环境下的安全将牺牲效率，PoW 机制需要消耗算力，对算力的浪费也被一些人士的观点所诟病。为了面向真实世界的高并发应用，区块链平台纷纷由 PoW 机制转型。

EOS 采取了已经在 Bitshares 上验证成熟可行的 DPOS（股份授权证明）共识机制。

BCOS 和 Fabric 均认可 PBFT 和 RAFT 的共识机制，采用插件化设计实现，通过修改系统配置，即可以在一个联盟链里使用不同的共识机制。

CITA 当前支持 Tendermint 算法，亦称 BFT 共识机制。Tendermint 算法是一个加密技术平台 Tendermint 提供的技术方案。

以太坊虽然目前仍是采用 PoW 机制运行，但在以太坊的技术路线里早已提出将向 PoS（权益证明）机制转变。而以太坊本身也设置了算力难度随着区块高度的增加递增，最终整个网络难以出块的机制，表现了转向 PoS 机制的坚定决心。在具体实施中，提出了 PoS+PoW 的混合方案，也是由 PoW 机制的转型。

MOCA 保持现有比特币和以太坊这种公有共享（PoW），便于全球铺开的方式，但是又满足由 PoS 带来的速度，可以做到每秒几千笔的交易处理水平。MOAC 提出一个子母链的概念，母链，按照目前的以太坊的 PoW 算法（用 GHOST 解决分歧），就是可以全球铺开，但是速度慢且时效长；子链，按照 PoS 算法，已经被小型网络验证，可以有比较高的速度。

2. 记账权分配：公有链“凭股份”，联盟链“靠指派”

以太坊和 EOS 采用股份证明为主的形式。作为开放的公有链网络，拥有代币量大的账户将拥有更多的记账权。然而这一机制容易带来代币持有量大的节点而独占记账权。为此，PoS 和 DPoS 机制采取了不同解决方式。

一般性 POS 机制根据节点持有的代币的量和时间（币天）分配记账权。币天越大，获取记账权的可能性越大。通过一次记账后币天销毁的模式，避免持有代币份额大的节点长期独占记账权的情况。以太坊设计中 PoS 机制——Casper，通过下注共识来形成共识结果。通过确保每一个节点只有在由所有节点组成的联盟中才能获得最大利益的方式抵御多数派攻击。

DPoS 机制通过选举的方式，委任 N 个节点（见证人）参与记账。见证人轮流出块，出块顺序随机，权利完全对等。

仅供非商业用途或交流学习使用

联盟链 BCOS、Fabric 支持 PBFT、RAFT 的共识机制，用户可以根据自己的需求指定需要采用的共识机制。在记账权分配和共识机制的形成中，按既定规则指派记账权：PBFT 共识机制将服务器节点分为主节点和从节点两类，主节点轮流担任，由主节点排序共识请求，从节点按照主节点的顺序执行请求；而 RAFT 则更进一步，Leader 独占记账权，Follower 按照 Leader 的指令同步账本。虽然 Leader 也由选举产生，但不同于 DPoS 机制，Leader 不用轮流出块，在一定时期内独占记账权。

3. 拜占庭容错：越开放越高

记账权越开放的平台，容错能力越高，可以接纳不可信节点和故障节点的能力也越高。

PoS 机制具有 49% 的容错能力。

EOS 采用的 DPoS 机制具有 10/21 的容错能力，约 47%，低于 49%。

而 PBFT 具有 m/3m+1 的容错能力，接近 1/3。

Tendermint 能够容忍 1/3 的拜占庭节点。RAFT 是非拜占庭问题中的共识算法，假设不存在不可信节点，不考虑拜占庭将军问题。其容错能力为 0%，只适合在十分可信的环境中运行。

4. 激励机制：公有链靠代币，联盟链靠场景

为了维护网络节点的规模和正常运行，公有链的共识机制建立在经济激励之上。

一般性 PoS 机制中，记账节点产生区块完成记账后，其币天将会清零，而获得与币天数量成一定比例的代币利息奖励。

DPoS 机制中，每次生成一个块时，会奖励该区块生成者代币。所创建的代币数量由所有区块生成者所公布的期望报酬的中位数决定。EOS 可能被配置为限制区块生成者所得奖励上限，使代币供应的年总增长不超过 5%。

而目前的 BCoS 等联盟链，没有直接提到关于记账节点经济激励的内容。记账节点参与共识主要出于区块链应用带来的成本节约，即靠区块链场景应用带来的收益，如微众银行运行的联合贷款备付金管理及对账平台，把此前联合贷款结算需要 T+1 的对账周期缩短到准实时，提高了运行效率，参与方都获得了链外的业务收益。

1.5.5　性能对比

1. 参与共识节点数越少，性能越好

弃用 PoW 机制之后，节省了解决数学难题的成本，但仍有参与共识的节点通讯效率和协商成本。根据参与共识（记账）的节点数排序为 POS>DPOS>PBFT>RAFT。

1）PoS 机制可由全部持有代币份额的节点参与记账。以太坊尚没有实际采用，出块时间不确定，但至少应该优于目前的 15 秒出块时间。

2）DPoS 机制由选出的 N 个节点参与记账，EOS 的 DPoS 机制设计的出块时间是 3 秒。

3）PBFT 在联盟链的有限节点数中，通过一致性协议达成共识。虽然节点数有限，但

仅供非商业用途或交流学习使用

也需要节点间两两通信完成一致性协议。

4）RAFT 的 Follower 直接按 Leader 的指示同步账本，同一时期记账节点只有一个，直接而高效。RAFT 和 PBFT 设计均能实现秒级出块时间。

2. 环境越可信，性能越好

公有链由于技术与治理的高度去中心化，考虑拜占庭问题会牺牲效率，并且也无法取得强一致性。企业级应用中用户具有更好的协调机制，通过授权机制，网络环境更可信，能够提升性能。RAFT 完全基于可信环境，Follower 按照 Leader 的指令同步账本，效率高。

3. 异步处理成亮点

CITA 的共识服务只负责交易排序，交易处理服务只负责对排好顺序的交易进行处理，共识过程可以先于交易处理完成，交易处理服务可以异步执行。异步交易处理技术使 CITA 具有更好的共识性能。

MOAC 用了异步调用智能合约，智能合约的启动到完成可以跨区块，不是限定在一个区块内完成，这样也导致了 MOAC 的吞吐速度要快很多。

1.5.6 隐私保护对比

1. 联盟链共识隔离

企业级区块链平台在隐私保护方面殊途同归。总的来说，基于假名的隐私保护，不能满足企业级的隐私保护需求。将交易信息发送到全网共识，由于节点之间业务可能存在竞争关系，不利于商业秘密的保护，而零知识证明以及同态加密，可以做到在不知道交易数据的情况下执行交易。但是这些技术被认为并不成熟，性能难以实用，安全性有待时间检验。因此，在隐私保护方面，联盟链将需要隐私保护的数据不发送到全网，仅在有限节点进行共识和确认，即共识隔离。可见，网络分区共识隐私保护可能成为可行技术条件下的主流。

1）CITA：交易局部执行

CITA 只将隐私交易的哈希值发送到全网，而隐私交易的完整数据只在相关节点保存。

具体方案是，隐私交易提交后，加密传送给拥有解密私钥的节点，同时交易哈希值被打包进入区块链。隐私交易数据只在拥有解密私钥的相关节点上保存，相关节点先解密再执行交易，交易数据不会发送给不相关节点，这就杜绝了任何信息泄漏的可能。

2）BCOS：隐私数据甄别处理

BCOS 对数据的安全和敏感级别进行甄别，仅把需要对全网确认共识的数据发送到联盟链上进行共享。这一点与 CITA 非常相似。

通过细粒度的权限控制，利益相关方有权利看到明文数据，其他参与者没有权利看到明文数据，但其必须可以对密文数据的真实性进行验证。（以上为 BCOS 备选方案）

3）Fabric：订阅隐私通道

Fabric 采用一种非常有特色的隐私保护方案。Fabric 仅为网络节点提供共识服务，而网

仅供非商业用途或交流学习使用

络节点之间并不维护同一个账本，通过订阅通道的方式与共识服务相连接。只有订阅同一个通道的节点可以维护和分享同一个账本，从而成一个个具有保密性的通讯链路。账本和账本之间批次隔离，形成了一个个隐私的共识通道，完全杜绝了信息泄漏的可能。

2. 公有链弱隐私

以太坊和 EOS 在隐私保护方面基于假名保护，没有做更多的延展。

1.5.7　智能合约对比

1. 虚拟机与支持语言

以太坊基于以太坊虚拟机 EVM，主要支持的开发语言有 Solidity、Serpent、LLL。

以太坊之后推出的区块链平台，将支持更多种用于智能合约开发的编程语言。

BCOS 平台计划在下一版本支持更高性能的虚拟机，支持更主流的开发语言，如 JVM 虚拟机和 Java 开发语言。

Fabric 支持的语言有 Go、Java、Nodejs。

而 EOS 采取虚拟机独立架构，脚本语言和虚拟机的实现将独立于 EOS 操作系统技术，任何开发语言或虚拟机都可以通过 API 与 EOS 集成在一起。EOS 计划支持 C++ 语言。

2. 灵活的业务场景执行器配置

不同于以太坊只提供 EVM 虚拟机，CITA 将共识服务和交易处理独立后，可以为不同的业务场景提供不同的执行器处理。交易在经过共识服务排序后，由交易路由分配到不同的执行器处理。通过灵活的视图配置，CITA 可以全面支持各种应用场景。

1.5.8　技术路线对比

通过在可信环境中采用 PBFT 等共识算法，区块链平台的性能较 PoW 机制下有了质的飞跃。而 CITA 提出的异步处理及微服务技术等，将进一步提升共识和交易处理的性能。另一方面，通过“共识隔离”，使得在技术条件不足的情况下实现了隐私保护。

但在面向真实商业环境的应用中，现有平台仍然在某些方面存在不足，需要重点研究。

1. 预言机管理

智能合约将应用于真实商业环境，而其数据也将来自真实环境。预言机提供的解决方案让区块链的智能合约获取现实世界的不确定数据信息，但这一链接链上与链外的环节存在众多无法信任的环节。

如何保证预言机读取的数据准确、如何保证预言机不受攻击和控制、如何在预言机发生错误的情况下调整智能合约的执行等问题，都将直接影响智能合约的实用性和扩展性。

BCOS 提出可信信息管理，即采用特定的共识机制对预言机提交信息的确定性做出判断，让信息知晓者在经济利益驱动下基于区块链数字身份提交现实世界的数据信息，并采用一定的惩罚机制，在一定程度上确保信息数据的确定性和正确性。

2. 孤链窘境

为了解决传统互联网世界的信息孤岛问题，区块链使用去中心化网络的结构，试图通过信息共享来解决数据孤岛的问题。然而，众多区块链应用的出现，区块链的链与链之间并不互通，使区块链也面临这一种“孤链”的窘境。这不符合区块链的初衷。

如何根据业务功能、隐私保护、数据隔离、性能容量扩展的需求等，在同一个区块链平台实施多链共存？如何在身份准入机制、信息标准、业务形态都不一致的区块链平台之间实施信息和业务交互？这些有望成为开发的重要方向。

1.5.9 经济模型对比

1. 以太坊经济模型

建立在以太坊（公链）上应用的每一个操作需要支付以太币，包括转账、智能合约中每一步操作，这样防止了恶意程序的攻击，同时将以太币作为对矿工的奖励。

以太坊具有交易系统的价值，以太币可通过以太坊进行二级市场交易。以太币的消耗（销毁）和增发（挖矿）使其成为一个流动的经济系统，围绕着这个经济系统，以太坊本身也可看作一个去中心化的公司。

以太坊 Ethereum 系统目前存在的问题如下：

1）以太坊的性能不能支撑普通商业应用。

2）Gas 费用高。

3）以太坊系统设计太复杂，导致它难以按计划切换 PoS，而且经常发生相关的安全性问题。

4）以太坊目前的 PoW 共识机制成本太高（占到新币价值的 11%）。

2. EOS 经济模型

EOS 经济模型中，比较合理地考虑到超级节点、持有 EOS 的普通用户和社区的三方利益。

区块生产者的利益如下：

1）区块生产者提供带宽、计算能力和存储，获得出块 Token 奖励，类似矿工。

2）Token 的每年增长比例不会超过 5%。

Token 持有人的利益如下：

1）一个账户持有所有 Token 发行总量的 1%，那么账号就具有使用 1% EOS 计算资源的能力。

2）如果你持有 Token 但不使用 EOS 计算资源，你可以将 Token 借给别人并获得收益。

3）企业可以在 EOS 上创建应用供用户使用，并为用户支付 EOS 费用，类似银行提供的免费转账。

4）只要你在 EOS 上有状态数据存储（将来会用到），则你必须保留一定数量的 Token，也就意味着 Token 被消耗。

社区的利益在于用户可以选出 3 个优秀社区应用，每年能够获得一定数量的 Token。

EOS 从经济上的主要思考点如下：

1）Token 代表使用权，使用系统资源并不消耗 Token。

2）可流通的 Token 会随着系统存储的状态越来越多而减少。

3）系统使用费用与 Token 当前发币价格无关，这解决了以太坊 Gas 价格的问题。

3. 瑞波经济模型

瑞波目前是专门为跨境的外汇汇款、支付、清缴等设计的一个系统。靠收取手续费赚钱。目前瑞波有 3 种提供跨境交易的模式，分别为 xCurrent、xRapid、xVia。

xCurrent 是由中间银行作为中转来完成交易的，xRapid 是用 xrp 来完成中间交易的，而 xVia 则是由网关作为中转完成交易的。

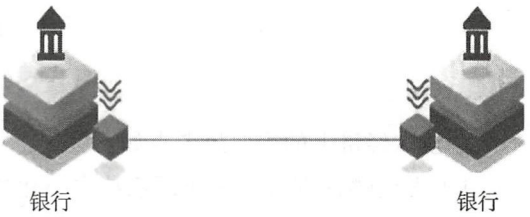


图 1-13　xCurrent 示意图

xCurrent 主要为银行与银行之间提供跨境交易，如图 1-13 所示。Ripple 网络在银行间设立了分布式的账本，每当银行 A 向银行 B 转账时，可以靠中间银行 C 进行清算。实质上通过分布式账本，使银行 A 在银行 C 开设的银行账户及银行 B 在银行 C 开设的银行账户内的金额发生了转变。这种模式的优点是速度快、费用低，缺点是 3 家银行都需要加入 Ripple 网络，并使用同一套分布式账本。

xRapid 模式为支付方先将支付金额换成 xrp，发送给收款方银行，银行将收到的 xrp 转换成当地货币，再支付给对应的收款方，如图 1-14 所示。这种模式比 xCurrent 更灵活，只需要收款方的银行可以接受 xrp 并换成当地货币即可。

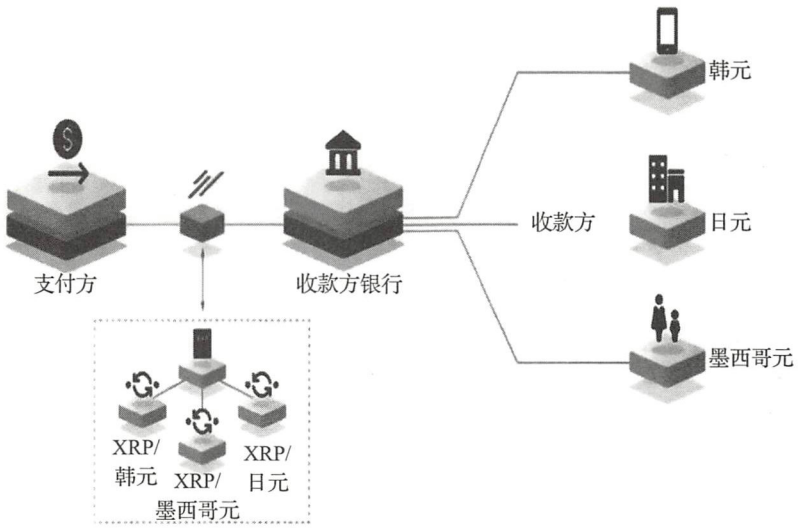


图 1-14　xRapid 示意图

xVia，是引入了网关的概念。网关是Ripple系统的一个中介机构（类似银行），Senders（支付方）可以将任意货币先转给网关，再由网关将货币转换成其他货币，支付给Beneficaries（收款人），如图 1-15 所示。这种模式最为灵活，支付方和收款方都不需要加入Ripple 网络，只需要信任网关即可。

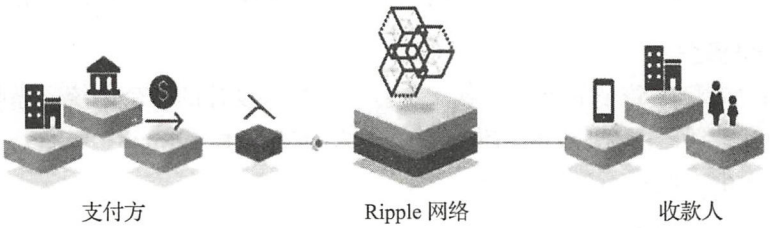
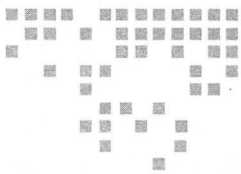


图 1-15　xVia 示意图

瑞波从经济上的主要思考点如下：

1）无论哪种方式，中间都依赖中心化机构（例如银行、网关等），才能完成整个环节。但是中心化机构就意味着安全性问题，尤其是网关，有可能会破产或者卷款逃跑。

2）xrp 币并不是必需的。而用 xCurrent 和 xVia 的模式，事实上并不一定要用到 xrp 币。



第 2 章 Chapter 2

分布式系统技术

区块链首先是一个分布式系统，区块链中的分布式包括以下几方面。

1）分布式的信息发布与传输：每个参与系统的节点都可以发起信息，每一个参与的节点可与相邻节点进行信息交互，全网公开传递有价值的信息。

2）分布式记账：每个参与系统的节点只要根据共识机制，完成工作量的设定，便能取得数据库的记账权，且记录可以追溯查询，但不可篡改。

3）分布式储存：分布式记账后在记录信息加工中加上一个时间戳，便会产生区块数据，经网络广播出去后，就会在区块链中形成一个账本，每个节点可以选择储存完整的数据或者部分数据。而且，每个节点都可以拥有一份完整的、实时更新的本地数据储存。

将区块链这种分布式的网络系统与中心化结构的网络系统对比，碰到的第一个问题就是一致性的保障。很显然，如果一个分布式集群无法保证处理结果一致的话，那任何建立于其上的业务系统都无法正常工作。

本章将介绍分布式系统中一些核心问题的来源以及相关的技术。

2.1 一致性问题

在分布式系统中，一致性（Consistency，早期也叫 Agreement）是指对于系统中的多个服务节点，给定一系列操作，在协议（往往通过某种共识算法）保障下，试图使得它们对处理结果达成某种程度的一致，即数据要完整、要同步。通常数据库中存在的“脏数据”是数据缺乏一致性的表现，而在分布式系统中常出现的不一致情况是由于读写数据时缺乏一致性，比如一个节点写操作之后没有对与其有数据冗余的另一个节点进行数据更新，这样在读

取另一个节点时就会出现数据不一致问题。

举个例子，某影视公司旗下有位于西单和中关村的两个电影院，都出售某部电影的电影票，一共 10000 张。那么，顾客到达其中一个电影院买票的时候，售票员该怎么决策是否卖出这张票又不会超售呢？当电影院个数更多的时候会怎样？这个问题在人类世界中，看起来似乎没那么难，但在分布式网络系统中，却面临不小的挑战。

有限状态机是有限个状态以及在这些状态之间的转移和动作等行为的数学模型，其特点是状态总数有限，任一时刻只处于一种状态中，而在某种条件下，会从一种状态转变到另一种状态。从有限状态机的角度来看一致性就是，各个节点构成相同的有限状态机，给定相同的初始状态和输入序列，保证在处理过程中每个环节的结果都相同。有限状态机示意图如图 2-1 所示。

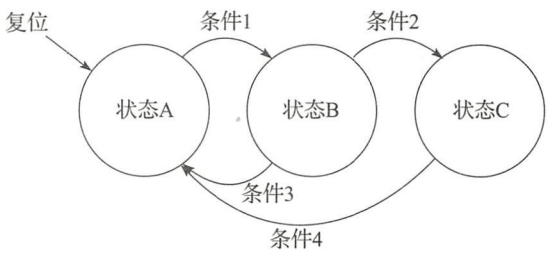


图 2-1 有限状态机示意图

在分布式环境里，要求多点数据具有一致性，如果分布式系统能实现“一致”，对外就可以呈现为一个功能正常的，且性能和稳定性都要好很多的“虚处理节点”，这也是分布式系统希望实现的最终目标。

2.1.1 问题挑战

在实际的计算机集群系统（看似强大的计算机系统的很多地方都比人类世界要脆弱得多）中，存在如下的问题：

1）节点之间的网络通信是不可靠的，包括任意延迟和内容故障。

2）节点的处理可能是错误的，甚至节点自身随时可能宕机。

3）同步调用会让系统变得不具备可扩展性。

要解决这些问题挑战，愿意动脑筋的读者可能会很快想出一些不错的思路。为了简化理解，仍然举两个电影院一起卖票的例子。可能有如下的解决思路：

1）每次要卖一张票前打电话给另外一家电影院，确认下当前票数并没超售。

2）两家电影院提前约好，奇数小时内一家可以卖票，偶数小时内另外一家可以卖票。

3）成立一个第三方存票机构，将票都放到那里，每次卖票向存票机构询问。

这些思路大致都是可行的。实际上，这些方法背后的思路是，将可能引发不一致的并行操作串行化，这就是现在计算机系统里处理分布式一致性问题的基础思路和唯一秘诀。只是因为计算机系统比较傻，需要考虑得更全面一些；而人们又希望计算机系统能工作得更快更稳定，所以算法需要设计得再精巧一些。

2.1.2 一致性的要求

规范的说，理想的分布式系统一致性应该满足以下要求。

1）可终止性（Termination）：一致的结果能在有限时间内完成。

2）共识性（Consensus）：不同节点最终完成决策的结果应该相同。

3）合法性（Validity）：决策的结果必须是其他进程提出的提案。

第 1 点很容易理解，这是计算机系统可以被使用的前提。需要注意，在现实生活中这点并不是总能得到保障的，例如取款机有时候会处于“服务中断”状态，电话有时候会“无法连通”。

第 2 点看似容易，但是隐藏了一些潜在信息。算法考虑的是任意的情形，凡事一旦推广到任意情形，就往往会有一些惊人的结果。例如现在就剩一张票了，中关村和西单的电影院也分别刚确认过这张票的存在，然后两个电影院同时来了一个顾客要买票，从各自“观察”看来，自己的顾客都是第一个到的……怎么能达成结果的共识呢？记住我们的唯一秘诀：核心在于需要把两件事情进行排序，而且这个顺序还得是大家都认可的。

第 3 点看似绕口，但是其实比较容易理解，即达成的结果必须是节点执行操作的结果。仍以卖票为例，如果两个影院各自卖出去 1000 张票，那么达成的结果就是还剩 8000 张，决不能认为票售光了。

2.1.3　一致性模型

本节讨论几种一致性模型，实现这几种一致性模型的难度是依次递减的，对一致性的要求强度也是依次递减的。

1. 强一致性（Strong Consistency）

做过分布式系统的读者应该能意识到，绝对理想的强一致性代价很大。除非不发生任何故障，所有节点之间的通信无须任何时间，这个时候其实就等价于一台机器了。实际上，越强的一致性要求往往意味着越弱的性能。强一致性的要求有两个：

1）任何一次读都能读到某个数据的最近一次写的数据。

2）系统中的所有进程看到的操作顺序，都和全局时钟下的顺序一致。

Maurice P.Herlihy 与 Jeannette M.Wing 在 1990 年撰写过一篇经典论文“Linearizability：A Correctness Condition for Concurrent Objects”，其中提出，在顺序一致性前提下加强了进程间的操作排序，形成唯一的全局顺序（系统等价于顺序执行，所有进程看到的所有操作的序列顺序都一致，并且跟实际发生顺序一致），是很强的原子性保证，但是比较难实现。这对全局时钟有非常高的要求。目前，高精度的石英钟的漂移率达到 10^{-8} 量级，人类目前最准确的原子震荡时钟的漂移率达到 10^{-11} 量级。分布式系统的各个进程、节点之间的时间并不能达成完美的一致性。Google 曾在其分布式数据库 Spanner 中采用基于原子时钟和 GPS 的 TrueTime 方案，能够将不同数据中心的时间偏差控制在 10ms 以内。方案简单粗暴而有效，但成本较高。

强一致的系统往往比较难实现。很多时候，人们发现实际需求并没有那么强，可以适当放宽一致性要求，降低系统实现的难度。例如，在一定约束下实现所谓最终一致性

（Eventual Consistency），即总会存在一个时刻（而不是立刻），系统达到一致的状态，这对于大部分的 Web 系统来说已经足够了。这一类弱化的一致性被笼统称为弱一致性（Weak Consistency），也就是顺序一致性。

2. 顺序一致性（Sequential Consistency）

顺序一致性也同样有两个条件，其一与前面强一致性的要求一样，也是可以马上读到最近写入的数据，然而它的第 2 个条件就弱化了很多，它允许系统中的所有进程形成自己合理的统一的一致性，不需要与全局时钟下的顺序一致。Leslie Lamport 在 1979 年撰写的经典论文“How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”中提出，顺序一致性保证所有进程看到的全局执行顺序（total order）一致，并且每个进程看自身的执行（local order）跟实际发生顺序一致。例如，某进程先执行 A，后执行 B，则实际得到的全局结果中就应该是 A 在 B 前面，而不能反过来。同时所有其他进程在全局上看到的也应该是这个顺序。顺序一致性实际上限制了各进程内指令的偏序关系，但不在进程间按照物理时间进行全局排序。

这里的第 2 个条件的要点在于：

1）系统的所有进程的顺序一致，而且是合理的，就是说任何一个进程中，这个进程对同一个变量的读写顺序要保持，然后大家形成一致。

2）不需要与全局时钟下的顺序一致。

可见，顺序一致性在顺序要求上并没有那么严格，它只要求系统中的所有进程达成自己认为的一致就可以了，即要错一起错，要对一起对，同时不违反程序的顺序即可，并不需要与全局顺序保持一致。

3. 因果一致性（Casual Consistency）

这在一致性的要求上，又比顺序一致性降低了：它仅要求有因果关系的操作顺序得到保证，非因果关系的操作顺序则无所谓。因果相关的要求如下。

1）本地顺序：本进程中，事件执行的顺序即为本地因果顺序。

2）异地顺序：如果读操作返回的是写操作的值，那么该写操作在顺序上一定在读操作之前。

3）闭包传递：和时钟向量里面定义的一样，如果 a->b，b->c，那么肯定也有 a->c。

用某社交网站朋友圈的“评论”为例说明，在 infoq 分享的朋友圈中，他们在设计数据一致性的时候，使用了因果一致性这个模型，用于保证对同一条朋友圈的回复的一致性。比如这样的情况：

A 发了朋友圈内容为梅里雪山的图片。

B 针对内容 A 回复了评论：“这里是哪里？”

C 针对 B 的评论进行了回复：“这里是梅里雪山”。

那么，这条朋友圈的显示中，显然 C 针对 B 的评论，应该在 B 的评论之后，这是一个

因果关系，而其他没有因果关系的数据则可以允许不一致。

2.2　一致性的共识算法

人们常常把一致性与共识混为一谈，实际上一致性描述的是结果状态，共识则是一种手段。一致性一般是指分布式系统中多个副本对外呈现的数据的状态，共识则描述了分布式系统中多个节点之间，对某个状态达成一致结果的过程。要保障系统满足不同程度的一致性，往往需要共识算法来达成。

共识算法解决的是对某个提案（Proposal）大家达成一致意见的过程。提案的含义在分布式系统中十分宽泛，如多个事件发生的顺序、某个键对应的值、谁是领导……可以认为，任何需要达成一致的信息都是一个提案。

2.2.1　问题挑战

如果分布式系统中各个节点都能保证以十分强大的性能（瞬间响应、高吞吐）无故障的运行，则实现共识过程并不复杂，简单通过多播过程投票即可。很可惜的是，现实中这样“完美”的系统并不存在，如响应请求往往存在时延、网络会发生中断、节点会发生故障，甚至存在恶意节点故意要破坏系统。

一般地，把故障（不响应）的情况称为“非拜占庭错误”，恶意响应的情况称为“拜占庭错误”(对应节点为拜占庭节点)。

2.2.2　常见算法

针对非拜占庭错误的情况，一般包括 Paxos、Raft 及其变种。

能容忍拜占庭错误的情况一般包括 PBFT 系列、PoW 系列算法等。从概率角度看，PBFT 系列算法是确定的，一旦达成共识就不可逆转；而 PoW 系列算法则是不确定的，随着时间推移，被推翻的概率越来越小。

1. Paxos 问题和算法

Paxos 问题是指分布式的系统中存在故障（fault），但不存在恶意（corrupt）节点场景（即可能消息丢失或重复，但无错误消息）下的共识达成（Consensus）问题。最早是 Leslie Lamport 用 Paxon 岛的故事模型来进行描述而命名。故事背景是古希腊 Paxon 岛上的多个法官在一个大厅内对一个议案进行表决，试图达成统一的结果。他们之间通过服务人员来传递纸条，但法官可能离开或进入大厅，服务人员可能偷懒去睡觉。

Paxos 算法是莱斯利·兰伯特（Leslie Lamport）于 1990 年提出的一种基于消息传递的一致性算法。Paxos 算法是一个解决分布式系统中多个节点之间就某个值达成一致的通信协议，能够保证在少数派离线的情况下，剩余的多数派节点仍然能够达成一致。Paxos 共识算

法从工程角度实现了一种最大化保障分布式系统一致性（存在极小的概率无法实现一致）的机制。Paxos 被广泛应用在 Chubby、ZooKeeper 这样的系统中，Leslie Lamport 因此获得了 2013 年度图灵奖。Paxos 是第一个被证明的共识算法，其原理基于两阶段提交并进行扩展。作为现在共识算法设计的鼻祖，他以最初论文的难懂（算法本身并不复杂）出名。

Paxos 算法中的 4 种角色如下：

1）client 议题产生者，产生一个待分布式系统达成一致的值 v。

2）proposer 提议者，用 client 产生的值 v，向 acceptor 发出提议。

3）acceptor 决策者，决定是否接受 proposer 的提议，若大多数接受了提议，则结果达成一致，且达成一致的结果不可更改。

4）learner 决策学习者，学习最终达成一致的结果。一旦学习成功，关闭对应的 paxos 过程（paxos instance），并通知 acceptor（或 acceptor 主动向 learner 获取）。

这 4 种角色中，proposer 和 acceptor 比较重要，协议主要的交互逻辑都在这两种角色中。

Paxos 算法的两阶段通信协议包含以下几个阶段：

（1）第一阶段 Prepare

client 产生一个值 v，并告知 proposer，我这里产生了一个待 accept 的值。

proposer 收到通知后，生成一个全局唯一并且递增的提案 ID，带着这个 ID（不需要携带 v）向集群中的所有 acceptor 发送 PrepareRequest 请求。acceptor 收到 PrepareRequest 请求后，检查一下之前接收到的提案 ID（包括第一阶段和第二阶段），新接收的提案 ID 用 n 表示，之前接收到的提案 ID 用 N 表示。如果 n <= N，返回拒绝，并携带 N 的值。如果 n>N，把 n 记录下来，以后不再接收提案 ID 比 n 小的提议，这时分两种情况：

1）之前没有 accept 任何值 v，返回可以接收提议；

2）之前已经 accept 过值，返回可以接收提议，并携带已经 accept，并且提案 ID 最大的值。

（2）第二阶段 Accept

如果 proposer 收到大多数 acceptor 的拒绝应答，回到第一阶段，根据接收到的最大的 N，把提案 ID 增大，继续发送 PrepareRequest。

如果 proposer 收到大多数 acceptor 可以接收提议的应答，从多个应答中选出提案 ID 最大的值（第一阶段如果 acceptor 已经 accept 过值，会返回提案 ID 最大的值），作为提案值。如果应答中没有值，选择 client 产生的值 v 作为提案值。然后携带当前的提案 ID，一起向集群中所有 acceptor 发送 AccpetRequest 请求。对这段话解释如下：第一阶段 client 产生的值 v，不一定作为 Accept 阶段的提案值。为了更快地达成一致，如果之前已经 accept 了值，那么 proposer 会倾向于把提案值修改为之前接受的值。各个 proposer 不是针锋相对的，而是合作共赢。acceptor 收到 AccpetRequest 后，检查请求中携带的提案 ID，如果此提案 ID 大于或等于 acceptor 记录的提案 ID（在第一阶段和第二阶段，acceptor 记录最大的提案 ID），接受提议并记录提案 ID 和提案值。否则拒绝，并返回记录的提案 ID。proposer 收到大多数

acceptor 接受提案的应答，形成决议，达成一致。

如果 proposer 收到大多数 acceptor 拒绝的应答，回到第一阶段，把提案增大（增加幅度依据 acceptor 返回的提案 ID），发送 PrepareRequest。

Paxos 是一致性协议的基础，其他的协议（raft、zab 等）都是 Paxos 的改进版本。Paxos 侧重理论，实现 Paxos 非常困难。谷歌 Chubby 论文中提到，从 Paxos 出发，在实现过程中处理了很多实际细节之后，已经变成另外一个算法了，这时候正确性无法得到理论的保证。所以后来才出现了许多基于 Paxos 的改进算法。

2. 拜占庭问题和算法

拜占庭问题更为广泛，讨论的是允许存在少数节点作恶（消息可能被伪造）场景下的一致性达成问题。拜占庭算法讨论的是最坏情况下的保障问题。

拜占庭将军（Byzantine Generals Problem）问题是 Leslie Lamport 于 1982 年提出的它是用来解释一致性问题的一个虚构模型，描述的是一个协议问题。拜占庭帝国军队的将军们必须全体达成一致，才能决定是否攻击某一支敌军。问题是这些将军在地理上是分隔开来的，并且将军中存在叛徒。叛徒可以任意行动以达到以下目标：欺骗某些将军采取进攻行动；促成一个不是所有将军都同意的决定，如当将军们不希望进攻时促成进攻行动；迷惑某些将军，使他们无法做出决定。如果叛徒达到了这些目的之一，则任何攻击行动的结果都是注定要失败的，只有完全达成一致的努力才能获得胜利。

拜占庭问题假设是对现实世界的模型化，由于硬件错误、网络拥塞或断开，甚至遭到恶意攻击，计算机和网络可能出现不可预料的行为。拜占庭容错协议必须处理这些失效，并且这些协议还要满足所要解决的问题要求的规范。这些算法通常以其弹性 t 作为特征，t 表示算法可以应付的错误进程数。很多经典算法问题只有在 $n \geqslant 3t+1$ 时才有解，如拜占庭将军问题，其中 n 是系统中进程的总数。

基于拜占庭将军问题，一致性的确保主要分为 3 个阶段：预准备（pre-prepare）、准备（prepare）和确认（commit）。流程如图 2-2 所示。

其中 C 为发送请求端，0、1、2、3 为服务端，3 为宕机的服务端。具体步骤如下。

1）request：请求端 C 发送请求到任意一节点，这里是 0。

2）pre-prepare：服务端 0 收到 C 的请求后进行广播，扩散至 1、2、3。

3）prepare：1、2、3 收到后记录并再次广播，1->0、2、3，2->0、1、3，服务端 3 因为宕机无法广播。

4）commit：0、1、2 节点在 prepare 阶段若收到超过一定数量的相同请求，则进入 commit 阶段，广播 commit 请求。

5）reply：0、1、2 节点在 commit

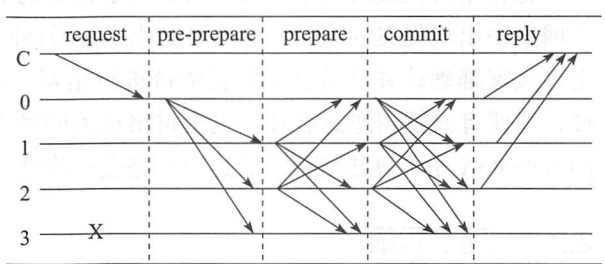


图 2-2　一致性协议流程图

阶段，若收到超过一定数量的相同请求，则对 C 进行反馈。

根据上述流程，在 $N \geqslant 3F+1$ 的情况下一致性是可能解决的。N 为总计算机数，F 为有问题的计算机总数。

$N=4$，$F=0$ 时：

	得到数据	最终数据
A	1 1 1 1	1
B	1 1 1 1	1
C	1 1 1 1	1
D	1 1 1 1	1

$N=4$，$F=1$ 时：

	得到数据	最终数据
A	1 1 1 0	1
B	1 1 0 1	1
C	1 0 1 1	1
D	0 1 1 1	1

$N=4$，$F=2$ 时：

	得到数据	最终数据
A	1 1 0 0	NA
B	1 0 0 1	NA
C	0 0 1 1	NA
D	0 1 1 0	NA

由此可以看出，拜占庭容错能够容纳将近 1/3 的错误节点误差。IBM 早期的区块链版本以及业界其他的区块链宣称是使用了该算法作为共识算法。拜占庭问题之所以难解，在于任何时候系统中都可能存在多个提案（因为提案成本很低），并且要完成最终的一致性确认过程十分困难，容易受干扰。但是一旦确认，即为最终确认。

比特币的区块链网络在设计时提出了创新的 PoW 算法思路。一个是限制一段时间内整个网络中出现提案的个数（增加提案成本），另外一个是放宽对最终一致性确认的需求，约定好大家都确认并沿着已知最长的链进行拓宽。系统的最终确认是概率意义上的存在。这样，即便有人试图恶意破坏，也会付出很大的经济代价（付出超过系统一半的算力）。后来的各种 PoX 系列算法也都沿着这个思路进行改进，采用经济上的惩罚来制约破坏者。

2.2.3 理论界限

搞学术的人都喜欢对问题先确定一个界限，那么，这个问题的最坏界限在哪里呢？很

不幸，一般情况下，分布式系统的共识问题无解。当节点之间的通信网络自身不可靠情况下，很显然，无法确保实现共识。但好在，一个设计得当的网络可以在大概率上实现可靠的通信。然而，即便在网络通信可靠情况下，一个可扩展的分布式系统的共识问题的下限是无解。

这个结论被称为 FlP 不可能性原理，可以看作分布式领域的“测不准原理”。

2.3　FlP 不可能原理

FlP 不可能原理：在网络可靠，存在节点失效（即便只有一个）的最小化异步模型系统中，不存在一个可以解决一致性问题的确定性算法。

提出该原理的论文由 Fischer、Lynch 和 Patterson 三位作者于 1985 年发表，该论文后来获得了 Dijkstra（就是发明最短路径算法的那位计算机科学家）奖。FlP 不可能原理实际上告诉人们，不要浪费时间去为异步分布式系统设计在任意场景下都能实现共识的算法。

理解这一原理的一个不严谨的例子如下：

3 个人在不同房间进行投票（投票结果是 0 或者 1）。3 个人彼此可以通过电话进行沟通，但经常会有人时不时地睡着。比如某个时候，A 投票 0，B 投票 1，C 收到了两人的投票，然后 C 睡着了。A 和 B 则永远无法在有限时间内获知最终的结果。如果可以重新投票，则类似情形每次在取得结果前发生。

FlP 原理实际上说明，在允许节点失效的情况下，纯粹的异步系统无法确保一致性在有限时间内完成。尽管 FlP 不可能原理是基于简单的系统模型假设，但我们可以根据归纳法得出，在更复杂的系统模型下，我们仍然没有任何算法能够完全保证分布式系统下的一致性。

这岂不意味着研究一致性问题根本没有意义吗？先别这么悲观，学术界做研究，考虑的是数学和物理意义上最极端的情形，很多时候现实生活要美好得多（感谢这个世界如此鲁棒！）。此结论只是说明了 100% 保证一致性是不可能的，这并不影响我们对分布一致性的探索（99% 以上的一致性还是完全有可能做到的）。例如，上面例子中描述的最坏情形总会发生的概率并没有那么大。工程实现上多试几次，会有很大成功的可能性。

科学告诉你什么是不可能的；工程则告诉你，付出一些代价，我可以把它变成可能。这就是工程的魅力。那么，退一步讲，在付出一些代价的情况下，我们能做到多少？回答这一问题的是另一个很出名的原理：CAP 原理。

2.4　CAP 原理

2.4.1　CAP 原理定义

分布式计算系统不可能同时确保一致性（Consistency）、可用性（Availability）和分区容忍性（Partition），设计中往往需要弱化对某个特性的保证。

一致性：任何操作应该都是原子的，发生在后面的事件能看到前面事件发生导致的结果（注意这里指的是强一致性）。

可用性：在有限时间内，任何非失败节点都能应答请求。

分区容忍性：网络可能发生分区，即节点之间的通信不可保障。

比较直观地理解，当网络可能出现分区时候，系统是无法同时保证一致性和可用性的。要么，节点收到请求后因为没有得到其他人的确认就不应答，要么节点只能应答非一致的结果。好在大部分时候网络被认为是可靠的，因此系统可以提供一致可靠的服务；当网络不可靠时，系统要么牺牲掉一致性（大部分时候都是如此），要么牺牲掉可用性。

CAP 原理最早由 Eric Brewer 于 2000 年在 ACM 组织的一个研讨会上提出猜想，后来 Lynch 等人进行了证明。该原理被认为是分布式系统领域的重要原理。

2.4.2 应用场景

既然 CAP 不可同时满足，则设计系统时候必然要弱化对某个特性的支持。

1. 弱化一致性

对结果一致性不敏感的应用，可以允许在新版本上线后过一段时间才更新成功，期间不保证一致性。例如，网站静态页面内容、实时性较弱的查询类数据库等，CouchDB、Cassandra 等为此设计。

2. 弱化可用性

对结果一致性很敏感的应用，例如银行取款机，当系统故障时候会拒绝服务。MongoDB、Redis 等为此设计。Paxos、Raft 等算法主要处理这种情况。

3. 弱化分区容忍性

现实中，网络分区出现概率减小，但较难避免。某些关系型数据库、ZooKeeper 即为此设计。实践中，网络通过双通道等机制增强可靠性，达到高稳定的网络通信。

比特币区块链依靠概率强一致性（Probabilistic strong consistency）实现一致性共识，这是中本聪提出来的。比特币大概每 10 分钟会分配出一个新的区块，这是靠很多机器不断地进行哈希计算产生出来的。如果两组机器算力相同，它们就会创建两个不同版本的新区块，然后就各自将这 10 分钟内自己知道的所有交易加入这个区块中，并且广播到自己周围。周围的节点在这区块以后再增加新的区块，形成了两条新的区块链，也就是分叉，各自有各自的树型结构。但是区块链内部机制有一个原则，总是选择最长的那条区块链，而丢弃其他的。这个机制其实非常类似现实自由选举制，选举人在各个州获得支持，最后获得更多人支持的获胜。

共识等待时间长度决定了是最终一致性还是强一致性，等待时间越长，一致性越高，通常等待比特币支付 24 小时后再发货是稳妥的办法。当区块链客户端在交易一加入区块链就立即接受，好处是不依赖其他节点，能够立即可用，是 CAP 的 AP，但是风险是可能丧失

强一致性，其他节点可能会丢弃这笔交易．因为其所在的区块链分叉在竞争性的选举中失败了。为了获得 CP，客户端应该等待区块链大多数节点接受了这笔交易后再真正接受它，说明这笔交易所在分叉已经取得选举胜利，获得大部分共识，获得了强一致性，但是风险是可能不可用，丧失 CAP 中的 A，因为网络分区通信等问题可能阻止这种共识。

2.5　ACID 原则

所谓 ACID 是数据库事务正确执行的 4 个基本要素的缩写，即：原子性（Atomicity）、一致性（Consistency）、独立性（Isolation）、持久性（Durability）。一个支持事务（Transaction）的数据库，必须具有这 4 种特性，否则在事务过程（Transaction processing）中无法保证数据的正确性，交易过程极可能达不到交易方的要求。ACID 原则描述了对分布式数据库的一致性需求，同时付出了可用性的代价。

1. 事务的原子性

事务的原子性指一个事务要么全部执行，要么不执行。也就是说，一个事务不可能只执行了一半就停止了。比如，你从取款机取钱，这个事务可以分成两个步骤：第 1 步划卡，第 2 步出钱。不可能划了卡，而钱却没出来。这两步必须同时完成，要么就全不完成。

区块链的原子性由共识机制、分叉理论和最长链原则共同控制。共识机制尽可能保证所有节点数据的原子性，但也会因为网络延迟和节点作恶导致部分节点更新失败。如果出现有的节点更新数据成功，有的节点更新数据失败的情况，则发生分叉，各节点根据最长链原则更新自己的数据。

2. 事务的一致性

事务的一致性指事务的运行并不改变数据库中数据的一致性。例如，完整性约束了 a+b = 10，一个事务改变了 a，那么 b 也应该随之改变。

区块链的一致性由共识机制控制各节点在一定时间窗口内“同步”更新，更新不成功即分叉。

3. 独立性

事务的独立性也称作隔离性，是指两个以上的事务不会出现交错执行的状态，因为这样可能会导致数据不一致。

区块链在共识机制控制下，单个时间点只有一个主体完成对区块链的更新操作，因此不存在并发事物操作，独立性得到较好的保障。

4. 持久性

事务的持久性是指事务执行成功以后，该事务对数据库所做的更改便持久地保存在数据库中，不会无缘无故地回滚。区块链的每个单独节点都是传统的数据库，因此能保证数据的持久性。

一个与之相对的原则是 BASE（Basic Availability，基本可用；Soft state，软状态；Eventually Consistency，最终一致性），牺牲掉对一致性的约束（最终一致性），来换取一定的可用性。

2.6 可靠性指标

很多领域一般都喜欢谈服务可靠性，用几个“9”来说事。这几个“9”其实是粗略代表了概率意义上系统能提供服务的可靠性指标，最初是电信领域提出的概念。表 2-1 给出不同指标下，每年允许服务出现不可用时间的参考值。

表 2-1 每年允许出现不可用时间的参考值

指标	概率可靠性	每年允许不可用时间	典型场景
1 个 9	90%	1.2 个月	不可用
2 个 9	99%	3.6 天	普通单点
3 个 9	99.9%	8.6 小时	普通企业
4 个 9	99.99%	51.6 分钟	高可用
5 个 9	99.999%	5 分钟	电信级
6 个 9	99.9999%	31 秒	极高要求
7 个 9	99.99999%	3 秒	N/A
8 个 9	99.999999%	0.3 秒	N/A
9 个 9	99.9999999%	30 毫秒	N/A

一般来说，单点的服务器系统至少应能满足 2 个 9；普通企业信息系统 3 个 9 就肯定足够了（大家可以统计下自己企业内因系统维护每年要停机多少时间），系统能达到 4 个 9 已经是业界领先水平了（参考 AWS）。电信级的应用一般号称能达到 5 个 9，这已经很厉害了，一年里面最多允许 5 分钟的服务停用。6 个 9 和以上的系统就更加少见了，要实现往往意味着极高的代价。

那么，该如何提升可靠性呢？有两个思路：一是让系统中的单点变得更可靠；二是消灭单点。IT 从业人员大都有类似的经验，运行某软系统的机器，基本上是过几天就要重启系统的；而运行 Linux 系统的服务器则可能几年时间都不出问题。另外，普通的家用计算机与专用服务器相比，长时间运行更容易出现故障。这些都是单点可靠性不同的例子。可以通过替换单点的软硬件来改善可靠性。

然而，依靠单点实现的可靠性毕竟是有限的，要想进一步可靠性提升，那就只好消灭单点，通过主从、多活等模式让多个节点集体完成原来单点的工作。这可以从概率意义上改善服务的可靠性，也是分布式系统的一个重要用途。

区块链的可靠性主要考察区块链网络、共享账本、账户体系 3 方面。

区块链网络的可靠性包括：记账节点高可用、服务节点之间高可用、区块链网络的网络

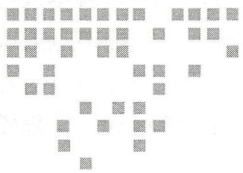
抖动是否影响系统服务等级。第一个和第二个指标的评测方法是通过脚本进行评测，设区块链中包含 n 个记账节点或服务节点，通过脚本依次停掉 1 ～ n 个记账节点或服务节点，检查区块链网络是否依然能够正确执行交易、达成共识并记账或通过服务节点对外提供服务；第三个指标的评测方法是通过工具进行模拟评测，在规定的服务等级和网络规模下，通过模拟一定量的网络节点加入或退出，获得其所容忍的比例值。

共享账本的可靠性包括：账本高可用、账本支持 failover 同步、账本备份恢复。第 1 个指标的测试方法是假设区块链中包含 n 个记账节点，每个记账节点均有一份共享账本，通过脚本依次停掉 1 ～ n 个记账节点，检查区块链网络是否依然能够正确执行交易、达成共识并记账，恢复被停掉的节点，检查是否能自动同步为最新账本；第 2 个指标的测试方法是通过脚本停掉区块链网络中的某一个记账节点，经过 10 个区块的生成时间后通过脚本重新开启该节点，恢复被停掉的节点，检查是否能自动同步为最新账本并恢复记账能力；第 3 个指标的测试方法是判断是否有备份和恢复的工具，若有，使用该工具进行备份导出操作，经过一段时间后，使用工具进行备份恢复导入操作，检查账本是否回到备份时的状态，并正常运行。

账户体系的可靠性主要是不同节点下的账户信息高可用。测试方法是利用脚本使某一账户所在节点失效，使用脚本对失效节点的账户进行模拟转账等交易行为，检查交易是否成功；若成功，则不同节点的账户信息高可用。

2.7　小结

分布式系统是计算机科学中十分重要的一个技术领域。常见的分布式一致性是个古老而重要的问题，无论在学术上还是工程上都有很高的价值。理想化（各项指标均最优）的解决方案是不存在的。在现实各种约束条件下，往往需要通过牺牲某些需求，来设计出满足特定场景的协议。其实，工程领域中很多问题的解决思路，都在于如何合理地进行取舍（trade-off）。



Chapter 3 第 3 章

密码学安全技术

公私钥密码算法是区块链系统的基石。在比特币大类的代码中，基本上使用的都是ECDSA。ECDSA 是 ECC 与 DSA 的结合，整个签名过程与 DSA 类似，所不一样的是签名中采取的算法为 ECC（椭圆曲线函数）。从技术上看，先从生成私钥开始，再从私钥生成公钥，最后从公钥生成地址。以上每一步都是不可逆的过程，也就是说无法从地址推导出公钥，也无法从公钥推导出私钥。

密码学相关的安全技术在区块链乃至整个信息技术领域的重要地位无须多言。如果没有现代密码学和信息安全的研究成果，人类社会根本无法进入信息时代。区块链技术大量依赖了密码学和安全技术的研究成果。实际上，密码学和安全领域所涉及的知识体系十分繁杂，本章将介绍密码学领域中跟区块链相关的一些基础知识，包括 Hash 算法与数字摘要、加密算法、数字签名、数字证书、PKI 体系、Merkle 树、布隆过滤器、同态加密等。读者通过阅读本章可以了解如何使用这些技术保护信息的机密性、完整性、认证性和不可抵赖性。

3.1 Hash 算法与数字摘要

在区块链领域，应用得最多的是哈希算法。哈希算法具有抗碰撞性、原像不可逆、难题验证友好性等特征。难题友好性正是众多 PoW 币种赖以存在的基础。在比特币中，SHA-256 算法被用作工作量证明的计算方法，也就是常说的挖矿算法。莱特币采用 Scrypt 算法，该算法与 SHA-256 不同的是，需要大内存支持。而在其他一些币种身上，也能看到基于 SHA-3 算法的挖矿算法。以太坊使用了 Dagger-Hashimoto 算法的改良版本，并命名为

Ethash，这是一个 IO 难解性的算法。除了挖矿算法，还会使用 RIPEMD160 算法，主要用于生成地址，众多的比特币衍生代码中，绝大部分都采用了比特币的地址设计。

3.1.1　Hash 定义

Hash（哈希或散列）算法是非常基础也非常重要的计算机算法，它能将任意长度的二进制明文串映射为较短的（通常是固定长度的）二进制串（Hash 值），并且不同的明文很难映射为相同的 Hash 值。例如计算一段话“ hello blockchain world，this is yeasy@github”的 SHA-256 Hash 值。计算结果如下：

```
$ echo "hello blockchain world, this is yeasy@github"|shasum -a 256
db8305d71a9f2f90a3e118a9b49a4c381d2b80cf7bcef81930f30ab1832a3c90
```

这意味着对于某个文件，无须查看其内容，只要其 SHA-256 Hash 计算后结果同样为 db8305d71a9f2f90a3e118a9b49a4c381d2b80cf7bcef81930f30ab1832a3c90，则说明文件内容极有可能就是“hello blockchain world，this is yeasy@github”。

Hash 值在应用中又常被称为指纹（fingerprint）或摘要（digest）。Hash 算法的核心思想也经常被应用到基于内容的编址或命名算法中。一个优秀的 Hash 算法将能实现如下功能。

1）正向快速：给定明文和 Hash 算法，在有限时间和有限资源内能计算得到 Hash 值。

2）逆向困难：给定（若干）Hash 值，在有限时间内很难（基本不可能）逆推出明文。

3）输入敏感：原始输入信息发生任何改变，新产生的 Hash 值都应该有很大不同。

4）冲突避免：很难找到两段内容不同的明文，使得它们的 Hash 值一致（发生碰撞）。

冲突避免有时候又称为“抗碰撞性”，分为“弱抗碰撞性”和“强抗碰撞性”。如果给定明文前提下，无法找到与之碰撞的其他明文，则算法具有“弱抗碰撞性”；如果无法找到任意两个发生 Hash 碰撞的明文，则称算法具有“强抗碰撞性”。

很多场景下，也往往要求算法对于任意长的输入内容，可以输出定长的 Hash 值结果。

3.1.2　常见算法

目前常见的 Hash 算法包括 MD5 和 SHA 系列算法。

MD4（RFC 1320）是 MIT 的 Ronald L.Rivest 在 1990 年设计的，MD 是 Message Digest 的缩写。其输出为 128 位。MD4 已被证明不够安全。

MD5（RFC 1321）是 Rivest 于 1991 年对 MD4 进行改进得到的。它对输入仍以 512 位进行分组，其输出是 128 位。MD5 比 MD4 更加安全，但过程更加复杂，计算速度要慢一点。MD5 已被证明不具备“强抗碰撞性”。

SHA（Secure Hash Algorithm）并非一个算法，而是一个 Hash 函数族。NIST（National Institute of Standards and Technology）于 1993 年发布其首个实现。目前知名的 SHA-1 算法在 1995 年面世，它的输出为长度 160 位的 Hash 值，抗穷举性更好。SHA-1 设计时模仿了 MD4 算法，采用了类似原理。SHA-1 已被证明不具备“强抗碰撞性”。

为了提高安全性，NIST 还设计出了 SHA-224、SHA-256、SHA-384 和 SHA-512 算法（统称为 SHA-2），与 SHA-1 算法原理类似。SHA-3 相关算法也已被提出。

目前，MD5 和 SHA-1 已经被破解，一般推荐至少使用 SHA-256 或更安全的算法。

3.1.3 性能

Hash 算法一般都是计算敏感型的。意味着计算资源是瓶颈，主频越高的 CPU 运行 Hash 算法的速度也越快。因此可以通过硬件加速来提升 Hash 计算的吞吐量。例如采用 FPGA 来计算 MD5 值，可以轻易达到数十 Gbps 的吞吐量。

也有一些 Hash 算法不是计算敏感型的。例如 Scrypt 算法，计算过程需要大量的内存资源，节点不能通过简单地增加更多 CPU 来获得 Hash 性能的提升。这样的 Hash 算法经常用在避免算力攻击的场景。

3.1.4 数字摘要

顾名思义，数字摘要是对数字内容进行 Hash 运算，获取唯一的摘要值来指代原始完整的数字内容。数字摘要是 Hash 算法最重要的一个用途。利用 Hash 函数的抗碰撞性特点，数字摘要可以解决确保内容未被篡改过的问题。

细心的读者可能会注意到，从网站下载软件或文件时，有时会提供一个相应的数字摘要值。用户下载原始文件后可以在本地自行计算摘要值，并与提供的摘要值进行比对，可检查文件内容是否被篡改过。

3.1.5 Hash 攻击与防护

Hash 算法并不是一种加密算法，不能用于对信息的保护。但 Hash 算法常用于对口令的保存。例如用户登录网站需要通过用户名和密码来进行验证。如果网站后台直接保存用户的口令明文，一旦数据库发生泄露后果不堪设想。大量用户倾向于在多个网站选用相同或关联的口令。

利用 Hash 的特性，后台可以仅保存口令的 Hash 值，这样每次只要比对 Hash 值一致，则说明输入的口令正确。即便数据库泄露了，也无法从 Hash 值还原回口令，只有进行穷举测试。

然而，有时用户设置口令的强度不够，只是一些常见的简单字符串，如 password、123456 等。有人专门搜集了这些常见口令，计算对应的 Hash 值，制作成字典。这样通过 Hash 值可以快速反查到原始口令。这一类型以空间换时间的攻击方法包括字典攻击和彩虹表攻击（只保存一条 Hash 链的首尾值，相对字典攻击可以节省存储空间）等。

为了防范这一类攻击，一般采用加“盐”（salt）的方法。即保存的不是口令明文的 Hash 值，而是口令明文再加上一段随机字符串（即“盐”）之后的 Hash 值。Hash 结果和“盐”分别存放在不同的地方，这样只要不是两者同时泄露，攻击者就很难破解了。

3.1.6　区块链中的 Hash 应用

Hash 在区块链中用处广泛，其一我们称之为 Hash 指针。Hash 指针是指该变量的值是通过实际数据计算出来的且指向实际的数据所在位置，即其既可以表示实际数据内容又可以表示实际数据的存储位置。Hash 指针在区块链中主要有两个应用，第 1 个就是构建区块链数据结构。了解区块链的读者应该知道，区块链数据结构由创世区块向后通过区块之间的指针进行连接，每个区块中都存储了前一个区块的 Hash 指针。这样的数据结构的好处在于，后面区块可以查找前面所有区块中的信息，且区块的 Hash 指针的计算包含了前面区块的信息，从而在一定程度上保证了区块链的不易篡改的特性。第 2 个应用是构建 Merkle Tree。Merkle Tree 的各个节点使用 Hash 指针进行构建。

Hash 在区块链的其他技术中也有所应用，例如：交易验证以及数字签名等。

3.2　加密算法

加解密算法是密码学的核心技术，从设计理念上可以分为两大基本类型，如表 3-1 所示。

表 3-1　加解密算法的类型

算法类型	特点	优势	缺陷	代表算法
对称加密	加解密的密钥相同	计算效率高，加密强度高	须提前共享密钥，易泄露	DES、3DES、AES、IDEA
非对称加密	加解密的密钥不相关	无须提前共享密钥	计算效率低，仍存在中间人攻击可能	RSA、EIGamal、椭圆曲线系列算法

3.2.1　加解密系统基本组成

现代加解密系统的基本组成一般包括：加解密算法、加密密钥、解密密钥。其中，加解密算法自身是固定不变的，并且一般是公开可见的；密钥则是最关键的信息，需要安全地保存起来，甚至通过特殊硬件进行保护。一般来说，对同一种算法，密钥需要按照特定算法在每次加密前随机生成，长度越长，加密强度越大。加解密的基本过程如图 3-1 所示。

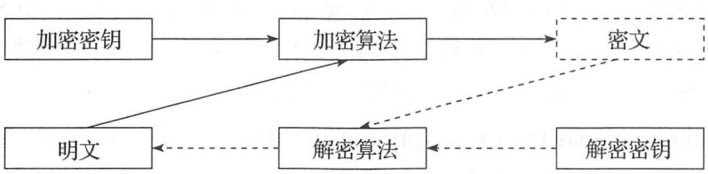


图 3-1　加解密的基本过程

加密过程中，通过加密算法和加密密钥，对明文进行加密，获得密文。

解密过程中，通过解密算法和解密密钥，对密文进行解密，获得明文。

根据加解密过程中所使用的密钥是否相同，算法可以分为对称加密（symmetric cryptography，又称公共密钥加密，common-key cryptography）和非对称加密（asymmetric cryptography，又称公钥加密，public-key cryptography）。两种模式适用于不同的需求，恰好形成互补。某些时候可以组合使用，形成混合加密机制。

并非所有加密算法的安全性都可以从数学上得到证明。公认的高强度的加密算法和实现往往需要经过长时间各方面充分实践论证后，才被大家所认可，但也不代表其绝对不存在漏洞。因此，使用自行设计和发明的、未经过大规模验证的加密算法是一种不太明智的行为。即便不公开算法加密过程，也很容易被攻破，在安全性上无法得到保障。

实际上，密码学实现的安全往往是通过算法所依赖的数学问题来提供的，而并非通过对算法的实现过程进行保密。

3.2.2 对称加密算法

对称加密算法，顾名思义，加密和解密过程的密钥是相同的。该类算法的优点是加解密效率（速度快，空间占用小）和加密强度都很高。缺点是参与方都需要提前持有密钥，一旦有人泄露则安全性被破坏；另外如何在不安全通道中提前分发密钥也是个问题，需要借助 Diffie-Hellman 协议或非对称加密方式来实现。

对称密码从实现原理上可以分为两种：分组密码和序列密码。前者将明文切分为定长数据块作为基本加密单位，应用最为广泛。后者则每次只对一个字节或字符进行加密处理，且密码不断变化，只用在一些特定领域，如数字媒介的加密等。

分组对称加密的代表算法有 DES、3DES、AES、IDEA 等。

1）DES（Data Encryption Standard）：是一种经典的分组加密算法。1977 年，美国联邦信息处理标准（FIPS）采用 FIPS-46-3，将 64 位明文加密为 64 位的密文，其密钥长度为 64 位（包含 8 位校验位）。现在已经很容易被暴力破解。

2）3DES：采取三重 DES 操作，加密→解密→加密，处理过程和加密强度优于 DES。但现在也被认为不够安全。

3）AES（Advanced Encryption Standard）：由美国国家标准研究所（NIST）采用，取代 DES 成为对称加密实现的标准。1997 年～ 2000 年 NIST 从 15 个候选算法中评选 Rijndael 算法（由比利时密码学家 Joan Daemon 和 Vincent Rijmen 发明）作为 AES，标准为 FIPS-197。AES 也是分组算法，分组长度有 128、192、256 位三种。AES 的优势在于处理速度快，整个过程可以用数学描述。目前尚无有效的破解手段。

4）IDEA（International Data Encryption Algorithm）：是 1991 年由密码学家 James Massey 与来学嘉联合提出的。其设计类似于 3DES，密钥长度增加到 128 位，具有更好的加密强度。

序列密码，又称流密码。1949 年，Claude Elwood Shannon（信息论创始人）首次证明，

要实现绝对安全的完善保密性（perfect secrecy），可以通过“一次性密码本”的对称加密处理。即通信双方每次使用与明文等长的随机密钥串对明文进行加密处理。序列密码采用了类似的思想，每次通过伪随机数生成器来生成伪随机密钥串。代表算法有 RC4 等。

对称加密算法适用于大量数据的加解密过程，但不能用于签名场景，并且往往需要提前分发好密钥。

3.2.3　非对称加密算法

非对称加密是现代密码学历史上一项伟大的发明，可以很好地解决对称加密算法中提前分发密钥的问题。

顾名思义，非对称加密算法中，加密密钥和解密密钥是不同的，分别称为公钥（public key）和私钥（private key）。私钥一般需要通过随机数算法生成，公钥可以根据私钥生成。公钥一般是公开的，他人可获取的；私钥一般是个人持有，他人不能获取。

非对称加密算法的优点是公私钥分开，在不安全通道中也可使用。缺点是处理速度（特别是生成密钥和解密过程）往往比较慢，一般比对称加解密算法慢 2 ～ 3 个数量级，同时加密强度也往往不如对称加密算法。

非对称加密算法的安全性需要基于数学问题来保障，目前主要有基于大数质因子分解、离散对数、椭圆曲线等经典数学难题进行保护。

代表算法包括：RSA、ElGamal、椭圆曲线、SM2 等。

1）RSA：是经典的公钥算法，1978 年由 Ron Rivest、Adi Shamir、Leonard Adleman 共同提出，他们于 2002 年因此获得图灵奖。算法利用了对大数进行质因子分解困难的特性，但目前还没有数学证明两者难度等价，或许存在未知算法可在不进行大数分解的前提下解密。

2）Diffie-Hellman 密钥交换：基于离散对数无法快速求解的特性，可以在不安全的通道上，由双方协商一个公共密钥。

3）ElGamal：由 Taher ElGamal 设计，利用了模运算下求离散对数困难的特性。被应用在 PGP 等安全工具中。

4）椭圆曲线算法（Elliptic Curve Cryptography，ECC）：是现代备受关注的算法系列，基于对椭圆曲线上特定点进行特殊乘法逆运算难以计算的特性。最早在 1985 年由 Neal Koblitz 和 Victor Miller 分别独立提出。ECC 系列算法一般被认为具备较高的安全性，但加解密计算过程往往比较费时。

5）SM2（ShangMi 2）：是我国商用密码算法，由国家密码管理局于 2010 年 12 月 17 日发布。它同样基于椭圆曲线算法，加密强度优于 RSA 系列算法。

非对称加密算法一般适用于签名场景或密钥协商，但不适于大量数据的加解密。

目前普遍认为，RSA 类算法可能在不远的将来被破解，一般推荐采用安全强度更高的椭圆曲线系列算法。

3.2.4 选择明文攻击

在非对称加密中，由于公钥是可以公开获取的，因此任何人都可以给定明文，获取对应的密文，这就带来选择明文攻击的风险。

为了避免这种风险，现有的非对称加密算法（如 RSA、ECC）都引入了一定的保护机制。对同样的明文使用同样密钥进行多次加密，得到的结果完全不同。这就避免了选择明文攻击的破坏。

在实现上可以有多种思路。一种是对明文先进行变形，添加随机的字符串或标记，再对添加后结果进行处理。另外一种是先用随机生成的临时密钥对明文进行对称加密，然后再进行对称密钥的加密，即混合利用多种加密机制。

3.2.5 混合加密机制

混合加密机制同时结合了对称加密和非对称加密的优点。

先用计算复杂度高的非对称加密协商出一个临时的对称加密密钥（也称为会话密钥，一般相对所加密内容来说要短得多），然后双方再通过对称加密算法对传递的大量数据进行快速的加解密处理。

典型的应用案例是现在大家常用的 HTTPS 协议。HTTPS 协议正在取代传统的不安全的 HTTP 协议，成为最普遍的 Web 通信协议。

HTTPS 在传统的 HTTP 层和 TCP 层之间通过引入 Transport Layer Security/Secure Socket Layer（TLS/SSL）加密层来实现可靠的传输。

SSL 协议最早是 Netscape 于 1994 年设计出来的、实现早期 HTTPS 的方案，SSL 3.0 及之前版本存在漏洞，被认为不够安全。TLS 协议是 IETF 基于 SSL 协议提出的安全标准，目前最新的版本为 1.2（2008 年发布）。推荐使用的版本号至少为 TLS 1.0，对应于 SSL 3.1 版本。除了 Web 服务外，TLS 协议也广泛应用于 E-mail、实时消息、音视频通话等领域。

采用 HTTPS 建立安全连接（TLS 握手协商过程）的基本步骤如图 3-2 所示。

1）客户端浏览器发送信息到服务器，包括随机数 R1、支持的加密算法类型、协议版本、压缩算法等。注意，该过程为明文。

2）服务端返回信息，包括随机数 R2、选定加密算法类型、协议版本以及服务器证书。注意，该过程为明文。

3）浏览器检查带有该网站公钥的证书。该证书需要由第三方 CA 来签发，浏览器和操作系统会预置权威 CA 的根证书。如果证书被篡改或作假（中间人攻击），很容易通过 CA 的证书验证出来。

4）如果证书没问题，则客户端用服务端证书中的公钥加密随机数 R3（又叫 Pre-MasterSecret），发送给服务器。此时，只有客户端和服务器都拥有 R1、R2 和 R3 信息，基于随机数 R1、R2 和 R3，双方才能通过伪随机数函数来生成共同的对称会话密钥 MasterSecret。

5）后续客户端和服务端的通信都通过对称加密算法（如 AES）进行保护。

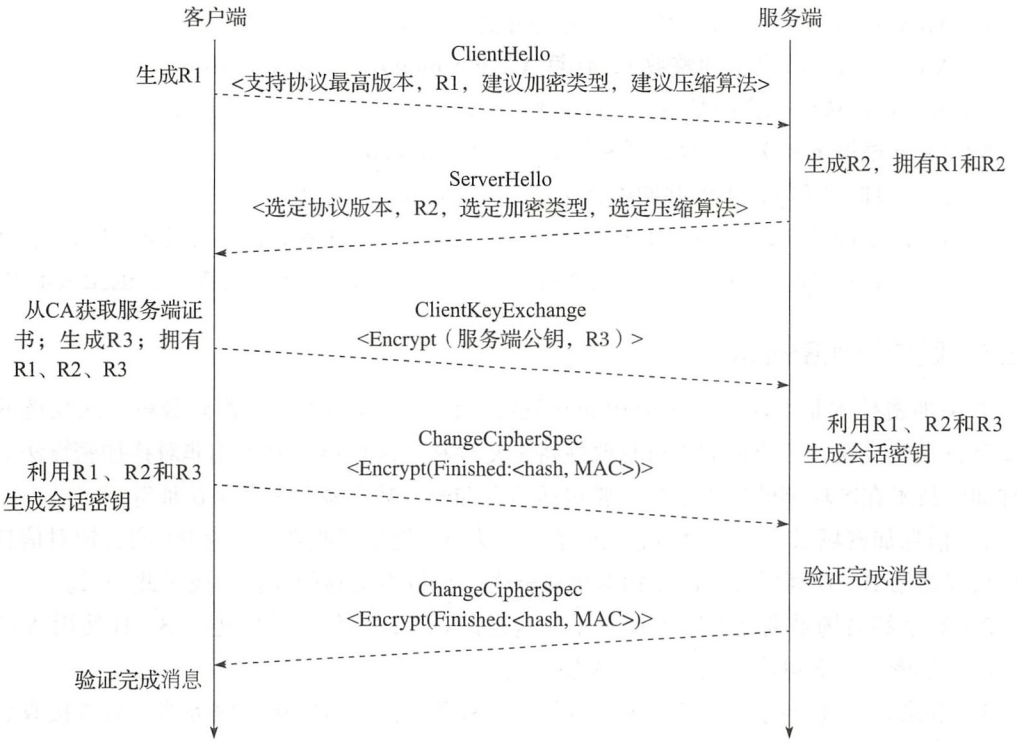


图 3-2　TLS 握手协商过程

可以看出，该过程的主要功能是在防止中间人窃听和篡改的前提下完成会话密钥的协商。为了保障前向安全性（perfect forward secrecy），TLS 对每个会话连接都可以生成不同的密钥，避免某次会话密钥泄露之后影响其他会话连接的安全性。需要注意，TLS 协商过程支持的加密算法方案较多，要合理地选择安全强度高的算法，如 DHE-RSA、ECDHE-RSA 和 ECDHE-ECDSA。

3.2.6　离散对数与 DH 密钥交换协议

上一小节中，对称密钥的协商过程采用了 RSA 非对称加密算法，实践中也可以通过 Diffie-Hellman 协议来完成。

DH（Diffie-Hellman）密钥交换协议是一个经典的协议，最早发表于 1976 年，应用十分广泛。使用该协议可以在不安全信道完成对称密钥的协商，以便后续通信采用对称加密。

DH 协议的设计基于离散对数问题（Discrete Logarithm Problem，DLP）。离散对数问题是指对于一个很大的素数 p，已知 g 为 p 的模循环群的原根，给定任意 x，求解 $X = g^x \bmod p$ 是可以很快获取的。但在已知 p、g 和 X 的前提下，逆向求解 x 目前没有多项式时间实现的算法。该问题同时也是 ECC 类加密算法的基础。

DH 协议的基本交换过程如下：

1）Alice 和 Bob 两个人协商密钥，先公开商定 p 和 g；

2）Alice 自行选取私密的整数 x，计算 $X = g\hat{}x \bmod p$，发送 X 给 Bob；

3）Bob 自行选取私密的整数 y，计算 $Y = g\hat{}y \bmod p$，发送 Y 给 Alice；

4）Alice 根据 x 和 Y，求解共同密钥 Z_A = $Y\hat{}x \bmod p$；

5）Bob 根据 X 和 y，求解共同密钥 Z_B = $X\hat{}y \bmod p$。

实际上，Alice 和 Bob 计算出来的结果将完全相同，因为在 mod p 的前提下，$Y\hat{}x = (g\hat{}y)\ \hat{}\ x = g\hat{}\ (xy) = (g\hat{}x)\ \hat{}y = X\hat{}y$。而信道监听者即便在已知 p、g、X、Y 的前提下，也无法求得 Z。

3.2.7 区块链加密技术

数字加密技术是区块链技术应用和开发的关键。一旦加密方法遭到破解，区块链的数据安全将受到挑战，区块链的不可篡改性将不复存在。区块链主要应用非对称加密算法。非对称加密技术在区块链的应用场景主要包括信息加密、数字签名和登录认证等。

1）信息加密场景主要是由信息发送者（记为 A）使用接收者（记为 B）的公钥对信息加密后再发送给 B，B 利用自己的私钥对信息解密。比特币交易的加密即属于此场景。

2）数字签名场景则是由发送者 A 采用自己的私钥加密信息后发送给 B，B 使用 A 的公钥对信息解密、从而可确保信息是由 A 发送的。

3）登录认证场景则是由客户端使用私钥加密登录信息后发送给服务器，后者接收后采用该客户端的公钥解密并认证登录信息 .

注意，上述 3 种场景加密的不同之处：

信息加密是公钥加密、私钥解密，确保信息的安全性；数字签名是私钥加密、公钥解密，确保数字签名的归属性；登录认证是私钥加密、公钥解密。

以比特币系统为例，其非对称加密机制如图 3-3 所示。

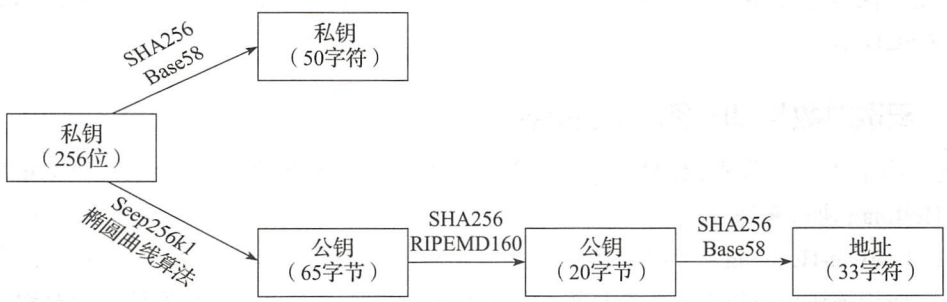


图 3-3 比特币非对称加密机制

比特币系统一般通过调用操作系统底层的随机数生成器来生成 256 位随机数作为私钥。比特币私钥的总量大，很难通过遍历全部私钥空间来获得存有比特币的私钥，因而在密码学上是安全的。为便于识别，256 位二进制形式的比特币私钥将通过 SHA-256 哈希算法和 Base-58 转换，形成 50 字符长度易识别和书写私钥提供给用户。比特币的公钥是由私钥首

先经过 Secp256k1 椭圆曲线算法生成 65 字节长度的随机数。该公钥可用于产生比特币交易时使用的地址，其生成过程是：首先将公钥进行 SHA-256 和 RIPEMD160 双哈希运算，生成 20 字节长度的摘要结果（即 Hash160 的结果），再经过 SHA-256 哈希算法和 Base-58 转换形成 33 字符长度的比特币地址。公钥生成过程是不可逆的，即不能通过公钥反推出私钥。比特币的公钥和私钥通常保存在比特币钱包文件中，其中私钥最为重要，丢失私钥就意味着丢失了对应地址的全部比特币资产。现有的比特币和区块链系统中，根据实际应用需求已经衍生出多私钥加密技术，以满足多重签名等更为灵活和复杂的场景。

下面介绍一下区块链项目 fabric 的加密算法。fabric 的加密算法由 BCCSP（Blockchain crypto provider，区块链加密服务提供商）提供，用于定义选择使用的密码学实现库。负责摘要生成、非对称密钥的签名与验证、根据证书查找私钥等。该模块提供了一系列的接口，这些接口定义了摘要的生成方法、签名、验证、加密、解密等。所有自定义的密码学实现库都需要实现这些接口，以达到密码学算法的可插拔。目前 fabric BCCSP 模块的接口有 3 种实现类，如图 3-4 所示。

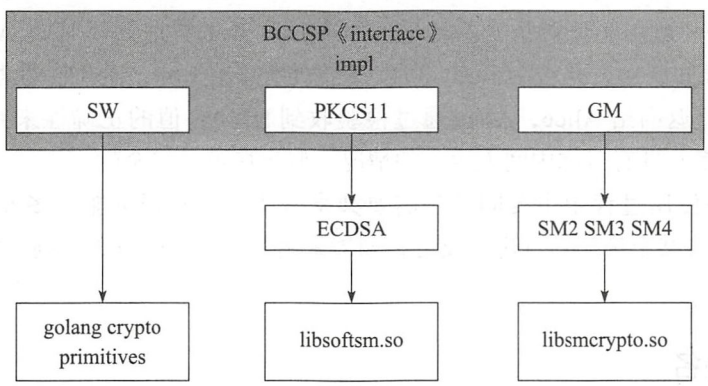


图 3-4　fabric BCCSP 模块接口的 3 种实现类

1）SW（software based）实现方式是直接调用 golang 提供的库文件来进行加密解密、哈希算法、签名验签等。

2）PKCS11 是调用 ECDSA 来进行加密、解密、哈希算法、签名验签等，而 ecdsa 是通过调用动态运行库来完成以上功能。

3）GM 国密算法实现方式与 PKCS11 一致。中间层提供 SM2 进行签名与验签，用 SM3 进行哈希算法，用 SM4 实现加密。

ECDSA：全名是 Elliptic Curve DSA，即椭圆曲线 DSA。它是数字签名算法 Digital Signature Algorithm，DSA）应用了椭圆曲线加密算法的变种。椭圆曲线算法的原理很复杂，但是具有很好的公开密钥算法特性，通过公钥无法逆向获得私钥。

PKCS：是由 RSA 实验室与本行业、学术界和政府的代表一起合作开发出的一套称为公共密钥加密标准（Public-Key Cryptography Standards）的规范。

3.3 消息认证码与数字签名

消息认证码和数字签名技术通过对消息的摘要进行加密，可用于消息防篡改和身份证明问题。

3.3.1 消息认证码

消息认证码的全称是“基于 Hash 的消息认证码”（Hash-based Message Authentication Code，HMAC）。消息验证码基于对称加密，可以用于保护消息完整性（integrity）。

基本过程为：对某个消息利用提前共享的对称密钥和 Hash 算法进行加密处理，得到 HMAC 值。该 HMAC 值持有方可以证明自己拥有共享的对称密钥，并且也可以利用 HMAC 确保消息内容不被篡改。

典型的 HMAC（K、H、Message）算法包括 3 个因素：K 为提前共享的对称密钥，H 为提前商定的 Hash 算法（一般为公认的经典算法如 SHA-256），Message 为要处理的消息内容。如果不知道 K 或 H 的任何一个，则无法根据 Message 得到正确的 HMAC 值。

消息认证码一般用于证明身份的场景。如 Alice、Bob 提前共享 HMCA 的密钥和 Hash 算法，Alice 需要知晓对方是否为 Bob，可发送随机消息给 Bob，Bob 收到消息后进行计算，把消息 HMAC 值返回给 Alice，Alice 通过检验收到 HMAC 值的正确性来判断对方是否是 Bob。注意，这里并没有考虑中间人攻击的情况，假定信道是安全的。

消息认证码使用过程中主要问题是需要共享密钥。当密钥可能被多方拥有的场景下，无法证明消息来自某个确切的身份。反之，如果采用非对称加密方式，则可以追溯到来源身份，即数字签名。

3.3.2 数字签名

数字签名基于非对称加密，与在纸质合同上签名确认合同内容和证明身份类似，既可以用于证实某数字内容的完整性，同时又可以确认来源（或不可抵赖，Non-Repudiation）。

一个典型的场景是，Alice 通过信道发给 Bob 一个文件（一份信息），Bob 如何获知所收到的文件是否 Alice 发出的原始版本？ Alice 可以先对文件内容进行摘要，然后用自己的私钥对摘要进行加密（签名），之后将文件和签名都发给 Bob。Bob 收到文件和签名后，用 Alice 的公钥来解密签名，得到摘要，与收到文件进行摘要后的结果进行比对。如果一致，说明该文件确实是 Alice 发过来的（别人无法拥有 Alice 的私钥），并且文件内容没有被修改过（摘要结果一致）。

知名的数字签名算法包括 DSA（Digital Signature Algorithm）和安全强度更高的 ECSDA（Elliptic Curve Digital Signature Algorithm）等。

除普通的数字签名应用场景外，针对一些特定的安全需求，产生了一些特殊数字签名技术，包括盲签名、多重签名、群签名、环签名等。

1）盲签名：盲签名（blind signature）是在1982年由David Chaum在论文“Blind Signatures for Untraceable Payment”中提出的。签名者需要在无法看到原始内容的前提下对信息进行签名。

一方面，盲签名可以实现对所签名内容的保护，防止签名者看到原始内容；另一方面，盲签名还可以实现防止追踪（unlinkability），签名者无法将签名内容和签名结果进行对应。典型的实现包括RSA盲签名算法等。

2）多重签名：多重签名（multiple signature）即在n个签名者中，收集到至少m个（$n >= m >= 1$）签名，即认为合法。其中，n是提供的公钥个数，m是需要匹配公钥的最少的签名个数。

多重签名可以被有效地应用在多人投票共同决策的场景中。例如双方进行协商，第三方作为审核方。三方中任何两方达成一致即可完成协商。

比特币交易中就支持多重签名，可以实现多个人共同管理某个账户的比特币交易。

3）群签名：群签名（group signature）即某个群组内一个成员可以代表群组进行匿名签名。签名可以验证来自于该群组，却无法准确追踪到签名的是哪个成员。

群签名最早于1991年由David Chaum和Eugene van Heyst提出。它需要存在一个群管理员来添加新的群成员，因此存在群管理员可能追踪到签名成员身份的风险。

4）环签名：环签名（ring signature）由Rivest、Shamir和Tauman三位密码学家在2001年首次提出。环签名属于一种简化的群签名。签名者首先选定一个临时的签名者集合，集合中包括签名者自身。然后签名者利用自己的私钥和签名集合中其他人的公钥就可以独立地产生签名，而无须他人的帮助。签名者集合中的其他成员可能并不知道自己被包含在最终的签名中。环签名在保护匿名性方面有很多的用途。

3.3.3 安全性

数字签名算法自身的安全性由数学问题进行保障，但在使用上，系统的安全性也十分关键。目前常见的数字签名算法往往需要选取合适的随机数作为配置参数，配置参数不合理的使用或泄露都会造成安全漏洞，需要进行安全保护。

2010年，SONY公司因为在其PS3产品上采用安全的ECDSA进行签名时，不慎采用了重复的随机参数，导致私钥被最终破解，造成重大经济损失。

3.3.4 区块链数字签名

数字签名涉及公钥、私钥和钱包等工具，它有两个作用：一是证明消息确实是由信息发送方签名并发出来的，二是确定消息的完整性。数字签名技术是将摘要信息用发送者的私钥加密，与原文一起传送给接收者。接收者只有用发送者的公钥才能解密被加密的摘要信息，然后用Hash函数对收到的原文产生一个摘要信息，与解密的摘要信息对比。如果相同，则

说明收到的信息是完整的，在传输过程中没有被修改，否则说明信息被修改过。因此数字签名能够验证信息的完整性。数字签名流程图如图 3-5 所示。

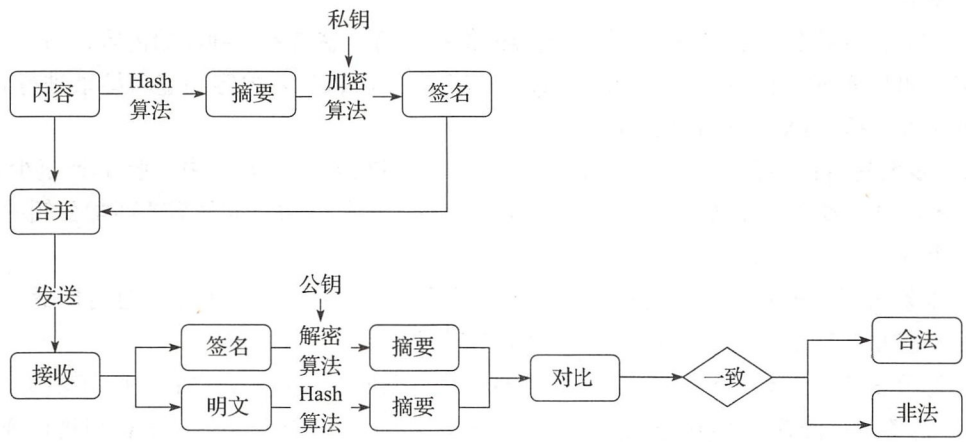


图 3-5 数字签名流程图

3.4 数字证书

对于非对称加密算法和数字签名来说，很重要的一点就是公钥的分发。理论上，任何人都可以公开获取对方的公钥。然而这个公钥有没有可能是伪造的呢？传输过程中有没有可能被篡改呢？一旦公钥自身出了问题，则整个建立在其上的安全体系的安全性将不复存在。

数字证书机制正好可以解决这个问题，它就像日常生活中的一个证书一样，可以证明记录信息的合法性。比如证明某个公钥是某个实体（如组织或个人）的，并且确保一旦内容被篡改能被探测出来，从而实现对用户公钥的安全分发。

根据所保护公钥的用途，数字证书可以分为加密数字证书（Encryption Certificate）和签名验证数字证书（Signature Certificate）。前者保护用于加密信息的公钥；后者则保护用于进行解密签名进行身份验证的公钥。两种类型的公钥也可以同时放在同一证书中。

一般情况下，证书需要由证书认证机构（Certification Authority，CA）来进行签发和背书。权威的证书认证机构包括 DigiCert、GlobalSign、VeriSign 等。用户也可以自行搭建本地 CA 系统，在私有网络中进行使用。

3.4.1 X.509 证书规范

一般来说，一个数字证书内容可能包括基本数据（版本、序列号）、所签名对象信息（签名算法类型、签发者信息、有效期、被签发人、签发的公开密钥）、CA 的数字签名等。

目前使用最广泛的标准为 ITU 和 ISO 联合制定的 X.509 的 v3 版本规范（RFC 5280），

其中定义了如下证书信息域。

- 版本号（Version Number）：规范的版本号，目前为版本 3，值为 0x2；
- 序列号（Serial Number）：由 CA 维护的、为它所颁发的每个证书分配的唯一的序列号，用来追踪和撤销证书。只要拥有签发者信息和序列号，就可以唯一标识一个证书，最大不能超过 20 字节；
- 签名算法（Signature Algorithm）：数字签名所采用的算法，如“sha256WithRSA-Encryption”或“ecdsa-with-SHA256”；
- 颁发者（Issuer）：颁发证书单位的标识信息，如“C = CN，ST = Beijing，L = Beijing，O = org.example.com，CN = ca.org.example.com”；
- 有效期（Validity）：证书的有效期限，包括起止时间；
- 主体（Subject）：证书拥有者的标识信息（Distinguished Name），如“C = CN，ST = Beijing，L = Beijing，CN = person.org.example.com”；
- 主体的公钥信息（Subject Public Key Info）：所保护的公钥相关的信息；
- 公钥算法（Public Key Algorithm）：公钥采用的算法；
- 主体公钥（Subject Public Key）：公钥的内容；
- 颁发者唯一号（Issuer Unique Identifier）：代表颁发者的唯一信息，仅 2、3 版本支持，可选；
- 主体唯一号（Subject Unique Identifier）：代表拥有证书实体的唯一信息，仅 2、3 版本支持，可选；
- 扩展（Extensions，可选）：可选的一些扩展，v3 中可能包括以下内容。

  ① Subject Key Identifier：实体的密钥标识符，区分实体的多对密钥；

  ② Basic Constraints：一般指明是否属于 CA；

  ③ Authority Key Identifier：证书颁发者的公钥标识符；

  ④ CRL Distribution Points：撤销文件的发布地址；

  ⑤ Key Usage：证书的用途或功能信息。

此外，证书的颁发者还需要对证书内容利用自己的公钥添加签名，以防止对证书内容进行篡改。

3.4.2　证书格式

X.509 规范中一般推荐使用 PEM（Privacy Enhanced Mail）格式来存储证书的相关文件。证书文件的文件名后缀一般为 .crt 或 .cer，对应私钥文件的文件名后缀一般为 .key，证书请求文件的文件名后缀为 .csr。有时候也统一用 .pem 作为文件名后缀。

PEM 格式采用文本方式进行存储，一般包括首尾标记和内容块，内容块采用 Base64 进行编码。

例如，一个 PEM 格式的示例证书文件如下：

```
-----BEGINCERTIFICATE-----MIICMzCCAdmgAwIBAgIQIhMiRzqkCljq3ZXnsl6EijAKBggqhkjO
PQQDAjBmMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNU2FuIEZyYW5jaX
XXbzEUMBIGA1UEChMLZXhhbXBsZS5jb20xFDASBgNVBAMTC2V4YW1wbGUuY29tMB4XDTE3MDQyNTAzMzAz
N1oXDTI3MDQyMzAzMzAzN1owZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBA
cTDVNhbiBGcmFuY2lzY28xFDASBgNVBAoTC2V4YW1wbGUuY29tMRQwEgYDVQQDEwtleGFtcGxlLmNvbTBZ
MBMGByqGSM49AgEGCCqGSM49AwEHA0IABCkIHZ3mJCEPbIbUdh/Kz3zWW1C9wxnZOwfyyrhr6aHwWREW3Z
pMWKUcbsYup5kbouBc2dvMFUgoPBoaFYJ9D0SjaTBnMA4GA1UdDwEB/wQEAwIBpjAZBgNVHSUEEjAQBgRV
HSUABggrBgEFBQcDATAPBgNVHRMBAf8EBTADAQH/MCkGA1UdDgQiBCBIA/DmemwTGibbGe8uWjt5hnlE63
SUsXuNKO9iGEhVqDAKBggqhkjOPQQDAgNIADBFAiEAyoMO2BAQ3c9gBJOk1oSyXP70XRk4dTwXMF7qR72i
jLECIFKLANpgWFoMoo3W91uzJeUmnbJJt8J1r00ByjurfAvv-----ENDCERTIFICATE-----
```

可以通过 OpenSSL 工具来查看其内容：

```
# openssl x509 -in example.com-cert.pem -noout -textCertificate:Data:Version:3
(0x2)SerialNumber:22:13:22:47:3a:a4:0a:58:ea:dd:95:e7:b2:5e:84:8aSignatureAlgo
rithm: ecdsa-with-SHA256Issuer:C=US, ST=California, L=SanFrancisco, O=example.
com,CN=example.comValidityNotBefore:Apr2503:30:372017GMTNotAfter:Apr2303:30:372
027GMTSubject:C=US, ST=California, L=SanFrancisco, O=example.com,CN=example.com
SubjectPublicKeyInfo:PublicKeyAlgorithm: id-ecPublicKeyPublic-Key: (256 bit)pub:0
4:29:08:1d:9d:e6:24:21:0f:6c:86:d4:76:1f:ca:cf:7c:d6:5b:50:bd:c3:19:d9:3b:07:f2:ca:
b8:6b:e9:a1:f0:59:11:16:dd:9a:4c:58:a5:1c:6e:c6:2e:a7:99:1b:a2:e0:5c:d9:db:cc:15:4
8:28:3c:1a:1a:15:82:7d:0f:44ASN1OID: prime256v1X509v3extensions:X509v3KeyUsage:
criticalDigitalSignature, KeyEncipherment, CertificateSign,CRLSignX509v3ExtendedKe
yUsage:AnyExtendedKeyUsage, TLSWebServerAuthenticationX509v3BasicConstraints: crit
icalCA:TRUEX509v3SubjectKeyIdentifier:48:03:F0:E6:7A:6C:13:1A:26:DB:19:EF:2E:5A:3B
:79:86:79:44:EB:74:94:B1:7B:8D:28:EF:62:18:48:55:A8SignatureAlgorithm: ecdsa-with-
SHA25630:45:02:21:00:ca:83:0e:d8:10:10:dd:cf:60:04:93:a4:d6:84:b2:5c:fe:f4:5d:19:3
8:75:3c:17:30:5e:ea:47:bd:a2:8c:b1:02:20:52:8b:00:da:60:58:5a:0c:a2:8d:d6:f7:5b:b3:
25:e5:26:9d:b2:49:b7:c2:65:af:4d:01:ca:3b:ab:7c:0b:ef
```

此外，还有 DER（Distinguished Encoding Rules）格式，它采用二进制来保存证书，可以与 PEM 格式互相转换。

3.4.3 证书信任链

证书中记录了大量信息，其中最重要的是“签发的公开密钥”和“CA 数字签名”两个信息。因此，只要使用 CA 的公钥再次对这个证书进行签名比对，就能证明某个实体的公钥是否合法。

读者可能会想到，怎么证明用来验证对实体证书进行签名的 CA 公钥自身是否合法呢？毕竟在获取 CA 公钥的过程中，它也可能被篡改。

实际上，要想知道 CA 的公钥是否合法，一方面可以通过更上层的 CA 颁发的证书来进行认证；另一方面某些根 CA（Root CA）可以通过预先分发证书来实现信任基础。例如，主流操作系统和浏览器里往往会提前预置一些权威 CA 的证书（通过自身的私钥签名，系统承认这些是合法的证书）。之后所有基于这些 CA 认证的中间层 CA（Intermediate CA）和后继 CA 都会被验证为合法。这样就由预先信任的根证书，经过中间层证书，到最底层的实体证书，构成一条完整的证书信任链。

某些时候用户在使用浏览器访问某些网站时，可能会被提示是否信任对方的证书。这说明该网站证书无法被当前系统中的证书信任链进行验证，需要进行额外检查。另外，若信任链上任一证书不可靠，则依赖它的所有后继证书都将失去保障。

可见，证书作为公钥信任的基础，对其生命周期进行安全管理十分关键。下节将介绍的 PKI 体系提供了一套完整的证书管理的框架，包括生成、颁发、撤销等。

3.5　PKI 体系

在非对称加密中，公钥可以通过证书机制来进行保护，但证书的生成、分发、撤销等过程并没有在 X.509 规范中进行定义。

实际上，安全地管理和分发证书可以遵循 PKI（Public Key Infrastructure）体系来完成。PKI 体系的核心是证书生命周期相关的认证和管理问题，在现代密码学应用领域处于十分重要的地位。

需要注意，PKI 是建立在公私钥基础上实现安全可靠传递消息和身份确认的一个通用框架，并不代表某个特定的密码学技术和流程。实现了 PKI 规范的平台可以安全可靠地管理网络中用户的密钥和证书。目前包括多个实现和规范，知名的有 RSA 公司的 PKCS（Public Key Cryptography Standards）标准和 X.509 相关规范等。

3.5.1　PKI 基本组件

一般情况下，PKI 至少包括如下核心组件。

- CA（Certification Authority）：负责证书的颁发和作废，接收来自 RA 的请求，是最核心的部分。
- RA（Registration Authority）：对用户身份进行验证，校验数据合法性，负责登记，审核过了就发给 CA。

证书数据库：存放证书，多采用 X.500 系列标准格式，可以配合 LDAP 目录服务管理用户信息。

其中，CA 是最核心的组件，主要完成对证书信息的维护。

常见的操作流程为，用户通过 RA 登记申请证书，提供身份和认证信息等；CA 审核后生成证书，颁发给用户。用户如果需要撤销证书则需要再次向 CA 发出申请。

3.5.2　证书的签发

CA 对用户签发证书实际上是对某个用户公钥使用 CA 的私钥对其进行签名。这样任何人都可以用 CA 的公钥对该证书合法性进行验证。验证成功则认可该证书中所提供的用户公钥内容，实现用户公钥的安全分发。

用户证书的签发可以有两种方式。一般可以由 CA 直接来生成证书（内含公钥）和对应

的私钥发给用户；也可以由用户自己生成公钥和私钥，然后由 CA 来对公钥内容进行签名。

后一种情况下，用户一般会首先自行生成一个私钥和证书申请文件（Certificate Signing Request，csr），该文件中包括了用户对应的公钥和一些基本信息，如通用名（common name，cn）、组织信息、地理位置等。CA 只需要对证书请求文件进行签名，生成证书文件，颁发给用户即可。整个过程中，用户可以保持私钥信息的私密性，不会被其他方获知（包括 CA 方）。

生成证书申请文件的过程并不复杂，用户可以很容易地使用开源软件 OpenSSL 来生成 csr 文件和对应的私钥文件。

例如，安装 OpenSSL 后可以执行如下命令来生成私钥和对应的证书请求文件：

```
$ openssl req -new -keyout private.key -out for_request.csr
Generating a 1024 bit RSA private key.....++++++.....++++++
writing new private key to 'private.key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:-----
You are about to be asked to enter information that will be incorporatedinto
your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,If you enter '.', the field will
be left blank.-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:Beijing
Locality Name (eg, city) []:Beijing
Organization Name (eg, company) [Internet Widgits Pty Ltd]:Blockchain
Organizational Unit Name (eg, section) []:Dev
Common Name (e.g. server FQDN or YOUR name) []:example.com
Email Address []:
Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

生成过程中需要输入地理位置、组织、通用名等信息。生成的私钥和 csr 文件默认以 PEM 格式存储，内容为 Base64 编码。

如生成的 csr 文件内容可能为：

```
$catfor_request.csr1-----BEGINCERTIFICATEREQUEST-----MIIBrzCCARgCAQAwbzELMAk
GA1UEBhMCQ04xEDAOBgNVBAgTB0JlaWppbmcxEDAOBgNVBAcTB0JlaWppbmcxEzARBgNVBAoTCkJsb2Nr
Y2hhaW4xDDAKBgNVBAsTA0RldjEZMBcGA1UEAxMQeWVhc3kuZ2l0aHViLmNvbTCBnzANBgkqhkiG9w0B
AQEFAAOBjQAwgYkCgYEA8fzVl7MJpFOuKRH+BWqJY0RPTQK4LB7fEgQFTIotO264ZlVJVbk8Yfl42F7d
h/8SgHqmGjPGZgDb3hhIJLoxSOI0vJweU9v6HiOVrFWE7BZEvhvEtP5klXXEzOewLvhLMNQpG0kBwdIh2
EcwmlZKcTSITJmdulEvoZXr/DHXnyUCAwEAAaAAMA0GCSqGSIb3DQEBBQUAA4GBAOtQDyJmfP64anQtR
uEZPZji/7G2+y3LbqWLQIcjIpZbexWJvORlyg+iEbIGno3Jcia7lKLih26lr04W/7DHn19J6Kb/CeXrj
DHhKGLOI7s4LuE+2YFSemzBVr4t/g24w9ZB4vKjN9X9i5hc6c6uQ45rNlQ8UK5nAByQ/TWDOxyG-----
ENDCERTIFICATEREQUEST-----
```

OpenSSL 工具提供了查看 PEM 格式文件明文的功能，如使用如下命令可以查看生成的 csr 文件的明文：

```
$ openssl req -in for_request.csr -noout -textCertificateRequest:Data:Version:
0 (0x0)Subject:C=CN, ST=Beijing, L=Beijing, O=Blockchain, OU=Dev,CN=yeasy.github.
comSubjectPublicKeyInfo:PublicKeyAlgorithm: rsaEncryptionRSAPublicKey: (1024 bit)
Modulus (1024 bit):00:f1:fc:d5:97:b3:09:a4:53:ae:29:11:fe:05:6a:89:63:44:4f:4d:02:
b8:2c:1e:df:12:04:05:4c:8a:2d:3b:6e:b8:66:55:49:55:b9:3c:61:f9:78:d8:5e:dd:87:ff:1
2:80:7a:a6:1a:33:c6:66:00:db:de:18:48:24:ba:31:48:e2:34:bc:9c:1e:53:db:fa:1e:23:95
:ac:55:84:ec:16:44:be:1b:c4:b4:fe:64:95:75:c4:cc:e7:b0:2e:f8:4b:30:d4:29:1b:49:01:
c1:d2:21:d8:47:30:9a:56:4a:71:34:88:4c:99:9d:ba:51:2f:a1:95:eb:fc:31:d7:9f:25Expon
ent:65537 (0x10001)Attributes:a0:00SignatureAlgorithm: sha1WithRSAEncryptioneb:50:
0f:22:66:7c:fe:b8:6a:74:2d:46:e1:19:3d:98:e2:ff:b1:b6:fb:2d:cb:6e:a5:8b:40:87:23:2
2:96:5b:7b:15:89:bc:e4:65:ca:0f:a2:11:b2:06:9e:8d:c9:72:26:bb:94:a2:e2:87:6e:a5:af
:4e:16:ff:b0:c7:9f:5f:49:e8:a6:ff:09:e5:eb:8c:31:e1:28:62:ce:23:bb:38:2e:e1:3e:d9:
81:52:7a:6c:c1:56:be:2d:fe:0d:b8:c3:d6:41:e2:f2:a3:37:d5:fd:8b:98:5c:e9:ce:ae:43:8
e:6b:36:54:3c:50:ae:67:00:1c:90:fd:35:83:3b:1c:86
```

需要注意，用户自行生成私钥情况下，私钥文件一旦丢失，CA 方由于不持有私钥信息，无法进行恢复，这意味着无法解密该证书中公钥加密的内容。

3.5.3　证书的撤销

证书超过有效期后会作废，用户也可以主动向 CA 申请撤销某证书文件。

由于 CA 无法强制收回已经颁发的数字证书，因此为了作废证书，往往还需要维护一个撤销证书列表（Certificate Revocation List，CRL），用于记录已经撤销的证书序号。

因此，在通常情况下，当第三方对某个证书进行验证时，需要首先检查该证书是否在撤销列表中。如果存在，则该证书无法通过验证；如果不在，则继续进行后续的证书验证过程。

3.6　Merkle 树结构

Merkle（默克尔）树，又叫哈希树，是一种典型的二叉树结构，由一个根节点、一组中间节点和一组叶节点组成。在区块链系统出现之前，它广泛用于文件系统和 P2P 系统中，如图 3-6 所示。

其主要特点为：最下面的叶节点包含存储数据或其哈希值；非叶子节点（包括中间节点和根节点）都是它的两个孩子节点内容的哈希值。

进一步地，默克尔树可以推广到多叉树的情形，此时非叶子节点的内容为它所有的孩子节点内容的哈希值。

默克尔树逐层记录哈希值的特点，让它具有了一些独特的性质。例如，底层数据的任何变动，都会传递到其父节点，一层层沿着路径一直到树根。这意味树根的值实际上代表了对底层所有数据的“数字摘要”。

仅供非商业用途或交流学习使用

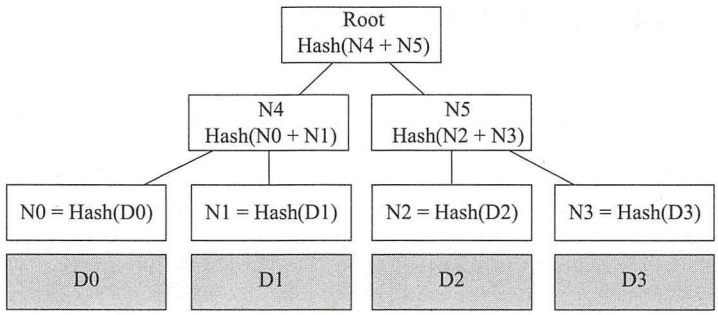


图 3-6 Merkle 树示例

目前，默克尔树的典型应用场景有很多，下面分别介绍。

3.6.1 快速对比大量数据

对每组数据排序后构建默克尔树结构。当两个默克尔树根相同时，则意味着两组数据必然相同；否则，必然存在不同。

由于 Hash 算法计算的过程可以十分快速，预处理可以在短时间内完成。利用默克尔树结构能带来巨大的对比性能优势。

3.6.2 快速定位修改

如图 3-5 所示，如果 D1 中数据被修改，会影响到 N1、N4 和 Root。因此，一旦发现某个节点如 Root 的数值发生变化，沿着 Root → N4 → N1，最多通过 O（logn）时间即可快速定位到实际发生改变的数据块 D1。

3.6.3 零知识证明

仍以图 3-5 为例，如何向他人证明拥有的某组数据（D0，…，D3）中包括给定某个内容 D0 而不暴露其他任何内容呢？很简单，构造如图 3-6 所示的一个默克尔树，公布 N1、N5、Root。D0 拥有者通过验证生成的 Root 是否跟提供的值一致，即可很容易检测 D0 存在。整个过程中验证者只知道 D0 以及公布的 N、N5、Root，验证的方法是沿着 D0、N1 得到 N4，再由 N4、N5 得到 Root, 验证计算得到 Root 和公布的 Root 是否一致即可。无法获知其他内容。

3.7 布隆过滤器

布隆过滤器（Bloom Filter）于 1970 年由 Burton Howard Bloom 在论文“Space/Time Trade-offs in Hash Coding with Allowable Errors”中提出。布隆过滤器是一种基于 Hash 的高效查找结构，能够快速（常数时间内）回答“某个元素是否在一个集合内”的问题。

布隆过滤器因为其高效性，大量应用于网络和安全领域，例如信息检索（BigTable 和 HBase）、垃圾邮件规则、注册管理等。

仅供非商业用途或交流学习使用

3.7.1　基于 Hash 值的快速查找

在运用布隆过滤器之前，先来看基于 Hash 的快速查找算法。在前面的讲解中我们提到，Hash 可以将任意内容映射到一个固定长度的字符串，而且不同内容映射到相同串的概率很低。因此，这就构成了一个很好的“内容→索引”的生成关系。

试想，如果给定一个内容和存储数组，通过构造 Hash 函数，让映射后的 Hash 值总不超过数组的大小，则可以实现快速的基于内容的查找。例如，内容“hello world”的 Hash 值如果是“100”，则存放到数组的第 100 个单元上去。如果需要快速查找任意内容，如“hello world”字符串是否在存储系统中，只需要在常数时间内计算其 Hash 值，并用 Hash 值查看系统中对应元素即可。该系统“完美地”实现了常数时间内的查找。

然而，令人遗憾的是，当映射后的值限制在一定范围（如总数组的大小）内时，会发现 Hash 冲突的概率会变高，而且范围越小，冲突概率越大。很多时候，存储系统的大小又不能无限扩展，这就造成算法效率的下降。为了提高空间利用率，后来人们基于 Hash 算法的思想设计出了布隆过滤器结构。

3.7.2　更高效的布隆过滤器

布隆过滤器采用了多个 Hash 函数来提高空间利用率。对同一个给定输入来说，多个 Hash 函数计算出多个地址，分别将位串的这些地址上标记为 1。查找时，进行同样的计算过程，并查看对应元素，如果都为 1，则说明该输入存在有较大概率，如图 3-7 所示。

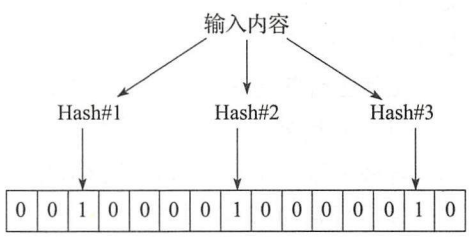


图 3-7　布隆过滤器

布隆过滤器查找相对于单个 Hash 算法查找，大大提高了空间利用率，可以使用较少的空间来表示较大集合的存在关系。

实际上，无论是 Hash 算法，还是布隆过滤器，基本思想是一致的，都是基于内容的编址。Hash 函数存在冲突，布隆过滤器也存在冲突。这就造成了两种方法都存在误报（false positive）的情况，但绝对不会漏报（false negative）。

布隆过滤器在应用中误报率往往很低，例如，在使用 7 个不同 Hash 函数的情况下，记录 100 万个数据，采用 2MB 大小的位串，整体的误判率将低于 1%。而传统的 Hash 查找算法的误报率将接近 10%。

3.8　同态加密

3.8.1　定义

同态加密（homomorphic encryption）是一种特殊的加密方法，允许对密文进行处理并

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

得到仍然是加密的结果。即对密文直接进行处理与对明文进行处理后再对处理结果加密，得到的结果相同。从抽象代数的角度讲，叫作保持了同态性。

同态加密可以保证实现处理者无法访问到数据自身的信息。如果定义一个运算符 Δ，对加密算法 E 和解密算法 D，满足：E(XΔY)=E(X)ΔE(Y)，则意味着对于该运算满足同态性。

同态性来自代数领域，一般包括 4 种类型：加法同态、乘法同态、减法同态和除法同态。同时满足加法同态和乘法同态，则意味着代数同态，称为全同态（full homomorphic）。同时满足 4 种同态性，则称为算数同态。对于计算机操作来讲，实现了全同态意味着对于所有处理都可以实现同态性。只能实现部分特定操作的同态性，称为特定同态（somewhat homomorphic）。

3.8.2 问题与挑战

同态加密的问题最早是由 Ron Rivest、Leonard Adleman 和 Michael L.Dertouzos 在 1978 年提出，同年提出了 RSA 加密算法。但第一个“全同态”的算法直到 2009 年才被克雷格·金特里（Craig Gentry）在论文“Fully Homomorphic Encryption Using Ideal Lattices”中提出并进行数学证明。仅满足加法同态的算法包括 Paillier 和 Benaloh 算法；仅满足乘法同态的算法包括 RSA 和 ElGamal 算法。同态加密在云计算和大数据的时代意义十分重大。目前，虽然云计算带来低成本、高性能和便捷性等优势，但从安全角度讲，用户还不敢将敏感信息直接放到第三方云上进行处理。有了比较实用的同态加密技术之后，大家就可以放心地使用各种云服务了，同时各种数据分析过程也不会泄露用户隐私。加密后的数据在第三方服务处理后得到加密后的结果，这个结果只有用户自身可以进行解密，整个过程中，第三方平台无法获知任何有效的数据信息。

对于区块链技术，同态加密也是很好的互补。使用同态加密技术，运行在区块链上的智能合约只能处理密文，而无法获知真实数据。这极大地提高了隐私安全性。

目前，全同态的加密方案主要包括如下 3 种类型。

1）基于理想格（ideal lattice）的方案：Gentry 和 Halevi 在 2011 年提出的基于理想格的方案可以实现 72 位的安全强度，对应的公钥大小约为 2.3GB，同时刷新密文的处理时间需要几十分钟；

2）基于整数上近似 GCD 问题的方案：Dijk 等人在 2010 年提出的方案（及后续方案）采用了更简化的概念模型，可以降低公钥大小至几十兆比特量级；

3）基于带扰动学习（Learning With Errors，LWE）问题的方案：Brakerski 和 Vaikuntanathan 等在 2011 年前后提出了相关方案；Lopez-Alt A 等在 2012 年设计出多密钥全同态加密方案，接近实时多方安全计算的需求。

目前，已知的同态加密技术往往需要较高的计算时间或存储成本，而且相比传统加密算法的性能和强度还有差距，但该领域被关注度一直很高。可以相信，在不远的将来会出现接近实用的方案。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

3.8.3　函数加密

与同态加密相关的一个问题是函数加密。同态加密保护的是数据本身，而函数加密保护的是处理函数本身，即让第三方在看不到处理过程的前提下，对数据进行处理。该问题已被证明不存在对多个通用函数的任意多密钥的方案，目前仅能做到对某个特定函数的一个密钥的方案。

3.9　其他问题

密码学领域涉及的问题还有许多，这里列出一些还在发展和探讨中的相关技术。

3.9.1　零知识证明概述

零知识证明（zero knowledge proof）是这样的一个过程，证明者在不向验证者提供任何额外信息的前提下，使验证者相信某个论断是正确的。

例如，Alice 向 Bob 证明自己知道某个数字，在证明过程中 Bob 可以按照某个顺序提出问题（比如数字加上某些随机数后的变换）由 Alice 回答，并通过回答确信 Alice 较大概率确实知道某数字。证明过程中，Bob 除了知道 Alice 确实知道该数字外，自己无法获知或推理出任何额外信息（包括该数字本身），也无法用 Alice 的证明去向别人证明（Alice 如果提前猜测出 Bob 问题的顺序，存在作假的可能性）。

零知识证明的研究始于 1985 年 Shafi Goldwasser 等人的论文“The Knowledge Complexity of Interactive Proof-Systems”，目前一般认为至少要满足以下 3 个条件。

1）完整性（Completeness）：真实的证明可以让验证者成功验证；

2）可靠性（Soundness）：虚假的证明无法让验证者保证通过验证，但允许存在小概率例外；

3）零知识（Zero-Knowledge）：如果得到证明，无法从证明过程中获知除了所证明信息之外的任何信息。

3.9.2　量子密码学

量子密码学（quantum cryptography）随着量子计算和量子通信的研究而受到越来越多的关注，将会对已有的密码学安全机制产生较大的影响。

量子计算的概念最早由物理学家费曼于 1981 年提出，基本原理是利用量子比特可以同时处于多个相干叠加态，理论上可以同时用少量量子比特来表达大量的信息，并同时进行处理，大大提高计算速度。如 1994 年提出的基于量子计算的 Shor 算法，理论上可以实现远超经典计算速度的大数因子分解。这意味着大量加密算法，包括 RSA、DES、椭圆曲线算法等都将很容易被破解。但量子计算目前离实际可用的通用计算机还有一定距离。

量子通信则提供对密钥进行安全分发的机制，有望实现无条件安全的“一次性密码”。量

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

子通信基于量子纠缠效应，两个发生纠缠的量子可以进行远距离的实时状态同步。一旦信道被窃听，则通信双方会获知该情况，丢弃此次传输的泄露信息。该性质十分适合于进行大量的密钥分发，如 1984 年提出的 BB84 协议，结合量子通道和公开信道，可以实现安全的密钥分发。

“一次性密码”最早由香农提出，实现理论上绝对安全的对称加密。其特点为密钥真随机且只使用一次；密钥长度与明文一致，加密过程为两者进行二进制异或操作。

3.9.3 社交工程学

密码学与安全问题，一直是学术界和工业界都十分关心的重要话题，相关的技术也一直在不断发展和完善。然而，即便存在理论上完美的技术，也不存在完美的系统。无数实例证实，看起来设计十分完善的系统最后被攻破，并非是因为设计上出现了深层次的漏洞，而问题往往出在事后看来十分浅显的某些方面。

例如，系统管理员将登录密码写在纸条上贴到计算机前；财务人员在电话里泄露用户的个人敏感信息；公司职员随意运行来自不明邮件的附件；不明人员借推销或调查问卷的名义进入办公场所窃取信息……

3.9.4 安全多方计算

安全多方计算用于解决一组互不信任的参与方之间保护隐私的协同计算问题。安全多方计算要确保输入的独立性、计算的正确性，同时不泄露输入值给参与方。通常，一个安全多方计算问题是指在一个分布式网络上计算基于任何输入的任何概率函数，每个输入方在这个分布式网络上都拥有一个输入，这个分布式网络要确保输入的独立性、计算的正确性，而且除了各自的输入外，不透露任何可用于推导其他输入和输出的信息。

可以将安全多方计算简单地概括为如下数学模型：在一个分布式网络中，有 n 个互不信任的参与者 P1, P2, ⋯, Pn，每个参与者 Pi 秘密输入 xi，他们需要共同执行函数 F:($x1$, $x2$, ⋯, xn) → ($y1$, $y2$, ⋯, yn)，其中 yi 为 Pi 得到的相应输出。在函数 F 的计算过程中，要求任意参与者 Pi（除 yi 外）均不能够得到其他参与者 Pi($j \neq i$) 的任何输入信息。

一般化的安全多方计算协议由于其计算任务的无关性（可以计算任意的功能函数），不需要再考虑特定的安全属性及外部运行环境，所以对现阶段复杂应用的安全保障具有得天独厚的优势。

3.10 小结

本章主要总结了密码学与安全领域中的一些核心问题和经典算法。

通过阅读本章内容，相信读者已经对现代密码学的发展状况和关键技术有了初步了解。掌握这些知识，对于理解区块链系统如何实现隐私保护和安全防护都很有好处。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

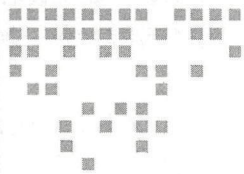
现代密码学安全技术在设计上大量应用了现代数学专业知识，如果读者希望成为这方面的专家，则需要进一步学习并深入掌握近现代的数学科学，特别是数论、抽象代数等相关内容。可以说，密码学安全学科是没有捷径可走的。

另外，从应用的角度来看，一套完整的安全系统除了核心算法外，还包括协议、机制、系统、人员等多个方面。任何一个环节出现漏洞都将带来巨大的安全风险。因此，要实现高安全可靠的系统是十分困难的。

区块链技术中大量利用了现代密码学的已有成果，包括哈希、加解密、签名、Merkle 树数据结构等。另一方面，区块链系统和诸多新的场景也对密码学和安全技术提出了很多新的需求，反过来也将促进相关学科的进一步发展。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用



Chapter 4 第 4 章

构建 Fabric 区块链网络

4.1 超级账本 Fabric 简介

超级账本（Hyperledger）是 Linux 基金会于 2015 年发起的推进区块链数字技术和交易验证的开源项目，加入成员包括：荷兰银行（ABN AMRO）、埃森哲（Accenture）等十几个不同利益体，目标是让成员共同合作，共建开放平台，满足来自多个不同行业各种用户案例，并简化业务流程。2016 年，IBM 与 Linux 进行了接触，共同创建了 IBM 的超级账本 Fabric 项目。为了帮助企业更轻松地构建区块链解决方案，Hyperledger 推出了一项开放计划：成立超级账本实验室，此举将是扩大超级账本社区的一种新途径。此前，超级账本需要企业证明自己的代码成熟度，并承诺提供资源，因此加入超级账本代码库的难度是非常大的。事实上，在目前近 200 个超级账本联盟成员当中，只有 8 个代码库获得了正式批准，而且其中也只有 3 个项目是“活跃状态”，分别是：英特尔锯齿湖项目（Sawtooth）、IBM Fabric 和日本 Soramitsu 公司的 Iroha。其中 IBM 的 Fabric 活跃程度是目前最高的，并且在联盟链业界的使用和研究最广泛。如果没有特别说明，本文后续的超级账本都特指 IBM 的超级账本 Fabric。

对比完全开放、去中心化和非授权的比特币及其他加密货币的区块链设计模型，超级账本是对该区块链模型的革新。即使希望授权的区块链作为起点，但超级账本通过提供一个模型，在某种程度上是允许创建授权的和非授权的区块链。另外，Fabric 通过一个提供针对身份识别、可审计和隐私的安全和健壮模型，使得缩短计算周期、提高规模效率和响应各个行业的应用需求成为可能。Fabric 提出了以下 3 个方面的愿景。

（1）区块链技术的未来愿景

我们相信，区块链技术将会对现实生活的很多方面产生根本性影响，从商业到数据存

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

储，以及其他事情。基于这一点，我们认为针对区块链 / 分布式账本技术制定健壮的和开放的标准是必要的，因为这能推动这样的技术在主流商业化领域得到应用。我们相信，未来世界将建立许多内部互联的分布式数据库和区块链，每一个分布式数据库和区块链都将满足特定用户的需求，但也要求与其他的账本进行通信。

（2）区块链技术的开放标准都必须尽量模块化

未来将使开发人员能够按照自己的意愿来回替换不同版本的各种区块链组件，必须建立这样的标准。例如，一些区块链应用案例在要求快速一致算法的同时要求更多的可信，而某些应用案例可能不要求速度但要求更加可信。密码算法、智能合约和数据库存储是需要实现“即插即用”的其他特征。模块化的另一个重要方面是使外部开发更容易。如果某个公司能够改进 Hyperledger 的某些模块，这个公司就可能做到并按自己的意愿发布这些改进的模块。实际上，公司或个人都应该能构建全部的模块（能够按要求一起使用，或者与其他 Hyperledger 组件实现“即插即用”），以容纳或与 Hyperledger 实现互动。本质上，不使用 Hyperledger 框架中的任何核心组件也能够构建区块链。Hyperledger 的长期愿景就是它包含丰富的、易用的 API 和数量庞大的核心模块，这样就能容易实现开发和互操作。虽然我们希望核心的 Hyperledger 模块能够满足尽量多的应用案例，但我们也知道 Hyperledger 的核心内容不可能覆盖每个行业的应用案例。然而，我们的 API 应该足够灵活，使得不使用 Hyperledger 核心组件构建的应用案例也能很容易地与核心的 Hyperledger 组件和区块链实现互动。

我们不可能考虑到 Hyperledger 和通用区块链技术将来所有的使用方式，因此，为了能够容纳将来未知的开发，Hyperledger 的设计应尽量模块化和可扩展。除此之外，Hyperledger 的模块化应该能让更多的人围绕 Hyperledger 工作。我们希望这种模块化的方式允许发明或开发新的区块链技术的人们发现：使用或与 Hyperledger 合作是很容易的。我们相信，针对任何区块链结构根本需求的一个方面就是网络中任何一方行为的识别和模式必须不能被未授权方通过检查账本就能查明。我们也期望某个需求允许区块链用户确认业务逻辑和 / 或交易机密的其他参数，使它们对任何人都是不可访问的，而不是合同的利益相关方或资产被转移。

（3）为核心协议之上轻松实现的各类丰富应用提供支持

这必将要求支持各种交易语义、密码算法、协商机制和数据库存储协议。例如，加密的 Hyperledger 应该包括所有的加密、签名和更高级的功能密码，从简单的、快速的对称加密到复杂的功能加密和基于属性的签名。这些基本的技术原理应通过配置来支持重要的商业交易，例如不同程度的授权交易的不可改变性和可审计性。我们希望 Hyperledger 是一个易用、十分有用和健壮的平台，任何对构建区块链软件的机构和个人都可以把它作为核心代码。尽管由于实际考虑不足，Hyperledger 针对每个潜在用户和应用案例可能缺乏这种理想的功能，但我们的目标就是使 Hyperledger 尽可能地接近这种理想的状态。

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

4.2 Fabric 特性和架构设计

4.2.1 Fabric 特性

本节主要讲述 Hyperledger Fabric 的关键设计特性，并简述如何实现了一个全面的、可定制的企业级区块链解决方案。

1. 资产定义

资产在这里理解为任何具有货币价值的东西，它们都可以通过网络进行交易，从超市商品到古董车再到货币期货都属于资产。资产可以包括有形资产（如房地产和硬件）和无形资产（如合同和知识产权）。Hyperledger Fabric 提供了使用智能合约交易修改资产的能力。资产在 Hyperledger Fabric 中以键 - 值对集合的形态存在，在通道（channel）中各本地账本可以对其状态提交变更事务。资产可以用二进制和 / 或 JSON 形式表示。我们可以通过 Hyperledger Fabric 的 Composer 工具很容易地定义和使用 Hyperledger Fabric 应用程序中的资产。

2. 智能合约

链码（chaincode）即 Fabric 的智能合约，分为系统链码和用户链码。链码的执行由事务排序划分，限制了节点类型间的信任和验证级别，并优化了网络的可伸缩性和性能。智能合约是定义资产并且可以用于修改资产的事务指令的软件。换句话说，它就是一个通道所有的业务逻辑。智能合约制定了执行读取或修改键 - 值对以及其他状态数据库信息操作的规则。智能合约通过一个事务请求来对账本的当前状态执行数据库操作。执行智能合约会生成一组读写集，这组读写集可以通过网络提交给排序服务节点，并由排序服务节点广播且应用到所有的对等节点上。

3. 账本特征

账本不可变地、共享地为每个通道编码了整个事务历史，还包含了类似 SQL 的查询功能，用于高效的审计和解决争议。在 Fabric 中产生的所有针对数据状态变更的请求都会生成有序且不可篡改的记录，并存于账本中。数据状态的变更是由所有参与方认可的智能合约调用事务的结果。每个事务都将产生一组资产键 - 值对，这些键 - 值对作为创建、更新或删除等操作而同步到所有账本。账本由区块链（区块根据 Hash 等算法组成的链条）组成，而每一个区块中都存储有一条或一组有序的且不可篡改的记录，也就是一个状态数据库来维护当前的 Fabric 的状态。每个通道都有且仅有一个账本，在该通道中的每个加盟成员的对等点都维护同一份账本。账本特征如下：

1）通过使用基键（键查询）、范围查询及组合键查询等方法可对账本执行查询和更新操作。

2）使用富查询语言的只读查询（可使用 CouchDB、LevelDB 作为状态数据库）。

3）只读历史查询——通过一个键来查询账本历史记录，支持数据来源场景。

仅供非商业用途或交流学习使用

4）每一条请求的结果都由通过智能合约读取的读集和智能合约写入的写集的键值的多版本组成。

5）每一条被提交的请求都包含提交该请求的节点的签名证书，并同时提交到排序服务节点。

6）同一个通道中的区块里的所有请求事务都会被排序，并且这些区块会被排序服务节点广播到该通道内的所有对等节点。

7）对等节点对请求事务的验证依靠背书策略，并严格执行该策略。

8）在添加一个块之前，先执行版本控制检查，以确保被读取的资产的状态在链代码执行时间之后没有改变。

9）即将执行变更的请求事务集在新增到一个区块之前必须要做一次版本验证，以确保被读取的资产状态集在本条智能合约执行时间之前没有改变过。

10）一旦请求事务被验证且提交，就不可篡改。

11）一个通道的账本包含一个区块生成的配置策略、访问控制列表和其他相关信息。

12）考虑到通道将会从不同的证书机构得到加密文件，因此通道中拥有成员服务提供者（MSP）实例。

4. 隐私通道

多通道交易的设计方案可以确保竞争的企业和受监管的行业在一个公共网络上交换资产时的高度隐私及保密性。Hyperledger Fabric 在每个通道中都有一个不可篡改的账本，以及一个可以操纵和修改当前资产状态的智能合约（例如，更新键 - 值对）。一个账本限制在一个通道的范围内——它可以在整个网络中共享（假设每个参与者都在一个公共通道上运行）——或者它也可以被私有化，只包含一组特定的参与者。在上述后一种情况下，这些参与者将创建一个单独的通道，从而使他们的事务和账本隔离出来。为了满足即公开透明又能保护隐私的场景，智能合约只能在需要访问资产状态来执行读和写操作的对等节点上安装（换句话说，如果一个智能合约没有安装在对等节点上，它将无法调用账本暴露出去的接口）。为了进一步混淆数据，智能合约中的值可以使用诸如 AES 之类的通用加密算法进行加密（在一定程度上或全部使用），然后将事务发送到排序服务，并将生成的区块追加到账本上。一旦加密的数据被写入账本，它只能被拥有相应密钥的用户解密。

5. 成员安全

Hyperledger Fabric 只允许被授权加盟的成员参与数据维护，且成员间相互认可，所有的交易都会被彼此发现和跟踪。这种方式提供了一个可信的区块链网络。Hyperledger Fabric 是一个支持所有参与者都有自己的身份的交易网络。公钥的底层方案用于生成与组织、网络组件和最终用户或客户端应用程序绑定的加密证书。因此，可以在更广泛的网络和通道层次上对数据访问控制进行操作和治理。隐私和保密是最重要和最关键的问题。Hyperledger Fabric 这种“被许可”的概念，再加上通道的存在和功能，有助于解决该问题。

仅供非商业用途或交流学习使用

6. 共识机制

共识策略的设定是为了达成一致的一种独特的方法，它可以实现企业间所需的灵活性和可伸缩性。在分布式账本技术中，共识作为单一功能最近成为一种特定算法的同义词。然而，共识不仅仅是简单地就事务的顺序达成一致，而且在 Hyperledger Fabric 中这种通过它在整个事务流中的基本作用，从提交请求、背书验证，到事务排序、确认和广播，这种区别显得尤为突出。简单地说，共识被定义为一个完整的循环，它是由一个经过验证核实的区块所包含的一组事务。当一个区块中的事务集合的顺序和结果经过所有的检查而符合策略标准时，将最终达成一致。这些检查和平衡发生在一个个请求事务的生命周期中，包括使用背书策略来规定哪些特定的成员必须支持某个事务类，以及系统智能合约，以确保这些策略得到执行和维护。在提交排序服务节点之前，这些执行验证的对等节点将使用这些系统智能合约来确保足够的背书支持，并从适当的实体中获得。此外，在任何包含事务的区块被添加进账本之前，将进行版本控制，在此期间，将对账本的当前状态进行商定或同意。最后的检查提供了对双重开销操作（处理相同事务）和其他可能危害数据完整性的威胁的保护，并允许被执行的方法依赖非静态变量。除了大量的背书认可、有效性和版本控制检查之外，在事务流的各个方向上也有正在进行的身份验证。访问控制列表是通过网络实现的（排序服务下发到所有通道），并且因为一条事务请求将会通过不同的体系结构组件，所以 payloads 会被反复地签名、复核及验证。总之，达成共识并不仅仅局限于一组交易的达成一致，而是一种包罗万象的特性，它是在交易从提案到最终广播过程中进行的持续验证的副产品。

4.2.2 Fabric 系统架构

超级账本 Fabric 项目自诞生之日起就吸引了全球众多企业的密切关注，已经先后发布了多个大的版本，0.6 实验版本（2016 年 9 月）、1.0 正式版本（2017 年 7 月）、1.1 版本（2018 年 4 月）。对比与 Fabric0.6 版本相似，1.0 版本的超级账本 Fabric 架构上核心特性如下。

（1）解耦节点的逻辑角色

Fabric 新版本解耦交易处理节点的逻辑角色为背书节点（Endorser）、确认节点（Committer），可以根据负载进行灵活部署，将原子排序环节与其他复杂处理环节解耦开，消除了网络处理瓶颈，提高了可扩展性。

（2）加强了身份证书管理

将身份认证管理服务作为单独的 Fabric CA 项目，提供更多功能。

（3）支持多通道

Fabric 新增加多通道特性，不同通道之间的数据彼此隔离，提高隔离安全性。

（4）支持可拔插的架构

将共识、权限管理、加解密、账本机制都模块化，支持多种类型。

（5）优化智能合约管理

Fabric 引入系统链码来实现区块链系统的处理，支持可编程和第三方实现。

仅供非商业用途或交流学习使用

超级账本 Fabric 的整体架构如图 4-1 所示。

图 4-1　Fabric 整体架构图

Fabric 为应用提供了 gRPC API，以及封装 API 的 SDK 供应用调用。应用可以通过 SDK 访问 Fabric 网络中的多种资源，包括账本、交易、事件、链码、权限管理等。应用开发者只需要跟这些资源打交道即可，而无须关心如何实现。其中，账本是最核心的结构，记录应用信息，应用则通过发起交易来向账本中记录数据。交易执行的逻辑通过链码来承载。整个网络运行中发生的事件可以被应用访问，以触发外部流程甚至其他系统。权限管理则负责整个过程中的访问控制。账本和交易进一步依赖核心的区块链数据结构、数据库、共识机制等技术；链码则依赖容器、状态机等技术；权限管理则利用了已有的 PKI 体系、数字证书、加解密算法等诸多安全技术。底层由多个节点组成 P2P 网络，通过 gRPC 通道进行交互，利用 Gossip 协议进行同步。层次化结构提高了架构的可扩展和可插拔性，方便开发者以模块为单位进行开发。

超级账本 Fabric 根据交易过程中不同环节的功能，在逻辑上将节点角色解耦为 Endorser 和 Committer，让不同类型节点可以关注处理不同类型的工作负载。典型的交易处理过程如图 4-2 所示。

图 4-2　Fabric 交易处理过程示例图

仅供非商业用途或交流学习使用

1. 组件功能

在整个交易过程中，各个组件的功能主要如下。

1）客户端（App）：客户端应用使用 SDK 与 Fabric 网络打交道。首先，客户端从 CA 获取合法的身份证书加入网络内的应用通道。发起正式交易前，需要先构造交易提案（Proposal）提交给 Endorser 进行背书（通过 EndorserClient 提供的 ProcessProposal(ctx context.Context, signedProp *pb.SignedProposal)(*pb.ProposalResponse, error) 接口）；客户端收集到足够（背书策略决定）的背书支持后，可以利用背书构造一个合法的交易请求，发给 Orders 进行排序（通过 BroadcastClient 提供的 Send(env *cb.Envelope)error 接口）处理。客户端还可以通过事件机制来监听网络中消息，以获知交易是否被成功接收。命令行客户端的主要实现代码在 peer/chaincode 目录下。

2）Endorser 节点：主要提供 ProcessProposal(ctx context.Context, signedProp *pb.SignedProposal)(*pb.ProposalResponse, error) 方法（代码在 core/endorser/endorser.go 中）供客户端调用，完成对交易提案的背书（目前主要是签名）处理。收到来自客户端的交易提案后，首先进行合法性和 ACL 权限检查，检查通过则模拟运行交易，对交易导致的状态变化（以读写集形式记录，包括所读状态的键和版本，所写状态的键值）进行背书并返回结果给客户端。注意，网络中可以只有部分节点担任 Endorser 角色。主要实现代码在 core/endorser 目录下。

3）Committer 节点：负责维护区块链和账本结构（包括状态 DB、历史 DB、索引 DB 等）。该节点会定期地从 Orders 获取排序后的批量交易区块结构，对这些交易进行落盘前的最终检查（包括交易消息结构、签名完整性、是否重复、读写集合版本是否匹配等）。检查通过后执行合法的交易，将结果写入账本，同时构造新的区块，更新区块中 BlockMetadata[2]（TRANSACTIONS_FILTER）记录交易是否合法等信息。同一个物理节点可以仅作为 Committer 角色运行，也可以同时担任 Endorser 和 Committer 这两种角色。主要实现代码在 core/committer 目录下。

4）Orders 节点：仅负责对交易排序。为网络中所有合法交易进行全局排序，并将一批排序后的交易组合生成区块结构。Orders 一般不需要跟账本和交易内容直接打交道。主要实现代码在 orders 目录下。对外主要提供 Broadcast(srv ab.AtomicBroadcast_BroadcastServer) error 和 Deliver(srv ab.AtomicBroadcast_DeliverServer)error 两个 RPC 方法。

5）CA：负责网络中所有证书的管理（分发、撤销等），实现标准的 PKI 架构。主要代码在单独的 fabric-ca 项目中。CA 在签发证书后，自身不参与网络中的交易过程。

2. 核心概念与组件

超级账本 Fabric 采用了模块化功能设计，整体的功能模块结构如图 4-3 所示。

超级账本 Fabric 面向不同的开发人员提供了不同层面的功能，自下而上可以分为 3 层。

1）网络层：面向系统管理人员。实现 P2P 网络，提供底层构建区块链网络的基本能力，包括代表不同角色的节点和服务。

2）共识机制和权限管理：面向联盟和组织的管理人员。基于网络层的连通，实现共识

仅供非商业用途或交流学习使用

机制和权限管理，提供分布式账本的基础。

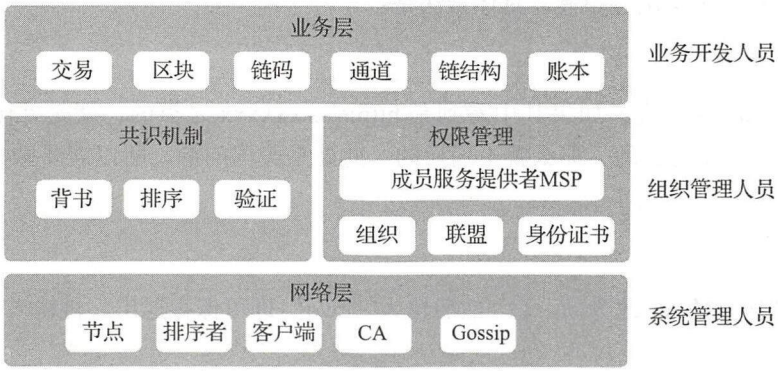


图 4-3　Fabric 功能模块图

3）业务层：面向业务应用开发人员。基于分布式账本，支持链码、交易等与业务相关的功能模块，提供更高一层的应用开发支持。

2018 年 4 月的 1.1 版本，相对于 1.0 版本，主要改进点如下：

1）支持新的 Node.js 类型 chaincode 和相关示例，以及在国内云环境落地的适配。

2）支持 chaincode 级别的账本数据加密新功能和相关示例，chaincode 支持对交易中的数据全程加密。

3）支持新的 connection profile 和自动生成功能，提供更为统一规范的应用连接配置，目前主要面向 Node.js 类型的 Client SDK 应用程序，未来有望推广到更多类型的 SDK 应用上。

4）支持基于 channel 的事件服务，允许客户端基于 channel 订阅块或块事务的事件。

5）链码支持 CouchDB 索引，提升系统性能。

6）支持生成证书吊销列表的能力。

7）支持动态更新与身份和从属关系的能力。

8）支持 Fabric 节点之间、节点和客户端之间的相互安全传输协议（TLS）。

9）链码支持基于属性的访问控制。

10）提供获取客户标识的链码 API，用于访问控制决策。

11）性能改进，提升事务吞吐量和响应效率。

12）内置容器化的 Hyperledger Blockchain Explorer。

4.3　Fabric 部署

4.3.1　单节点部署

1. Ubuntu 的安装

使用 VirtualBox 并在其中安装好 Ubuntu，下载最新版的 VirtualBox 和 Ubuntu Server(此

处用的是 Ubuntu16.04.2 X64 Server)。在安装完 Ubuntu 后，需要保证 apt source 是国内的，如果是国外的话速度会很慢很慢。具体做法是：

```
sudo vi /etc/apt/sources.list
```

打开这个 apt 源列表，如果其中看到是 http://us.xxxxxx 之类的，就是外国的；如果看到是 http://cn.xxxxx 之类的，那么就不用换的。此处是美国的源，所以需要做一下批量的替换。在命令模式下，输入

```
:%s/us./cn./g
```

就可以把所有的 us. 改为 cn. 了。然后输入“:wq”即可保存退出。再输入

```
sudo apt-get update
```

更新一下源。

然后安装 ssh。这样接下来就可以用 putty 或者 SecureCRT 之类的客户端远程连接 Ubuntu 了。

```
sudo apt-get install ssh
```

2. Go 的安装

Ubuntu 的 apt-get 虽然提供了 Go 的安装，但是版本比较旧，建议参考官方网站 https://golang.org/dl/，下载最新版的 Go，如图 4-4 所示。

图 4-4 下载最新版本的 Go

具体涉及的命令包括：

```
wget https://storage.googleapis.com/golang/go1.9.2 linux-amd64.tar.gz
```

将下载好的压缩包上传到服务器自定义目录：/root/zjlubuntu。
执行以下命令，解压压缩包：

```
sudotar -C /usr/local -xzf go1.9.2linux-amd64.tar.gz
```

注意：不要使用 apt 方式安装 Go，apt 的 Go 版本太低了！
接下来编辑当前用户的环境变量：

```
vi ~/.profile
```

添加以下内容：

```
export PATH=$PATH:/usr/local/go/bin
export GOROOT=/usr/local/go
export GOPATH=$HOME/go
export PATH=$PATH:$HOME/go/bin
```

编辑保存并退出 vi 后，记得载入以下环境：

```
source ~/.profile
```

我们把 go 的目录 GOPATH 设置为当前用户的文件夹下，所以记得创建 go 文件夹

```
cd ~
mkdir go
```

3. Docker 安装

我们可以使用阿里云提供的镜像，通过以下命令来安装 Docker，安装非常方便。

```
curl -sSL http://acs-public-mirror.oss-cn-hangzhou.aliyuncs.com/docker-engine/
internet | sh -
```

安装完成后需要修改当前用户（此处使用的用户名为 fabric）权限：

```
sudo usermod -aG docker fabric
```

注销并重新登录，然后添加阿里云的 Docker Hub 镜像：

```
sudo mkdir -p /etc/docker
sudo tee /etc/docker/daemon.json <<-'EOF'
{
"registry-mirrors": ["https://obou6wyb.mirror.aliyuncs.com"]
}
EOF
sudo systemctl daemon-reload
sudo systemctl restart docker
```

不同的版本添加方法是不一样的，官方的文档如下：

```
https://cr.console.aliyun.com/#/accelerator
```

喜欢用 DaoClound 的也可以用 DaoClound 的镜像。DaoCloud 的镜像设置文档为：

```
https://www.daocloud.io/mirror#accelerator-doc
```

4. Docker-Compose 的安装

Docker-Compose 是支持通过模板脚本批量创建 Docker 容器的一个组件。在安装 Docker-Compose 之前，需要安装 Python-pip。运行以下脚本：

```
sudo apt-get install python-pip
```

然后安装 docker-compose，可以从官方网站（https://github.com/docker/compose/releases）下载，也可以从国内的 DaoClound 下载。为了速度快，下面从 DaoClound 安装 Docker-Compose。运行以下脚本：

```
curl -L https://get.daocloud.io/docker/compose/releases/download/1.12.0/docker-compose-`uname -s`-`uname -m` > ~/docker-compose
sudo mv ~/docker-compose /usr/local/bin/docker-compose
chmod +x /usr/local/bin/docker-compose
```

5. Fabric 源码下载

我们可以使用 git 命令下载源码。首先需要建立对应的目录，然后进入该目录，git 下载源码。

```
mkdir -p ~/go/src/github.com/hyperledger
cd ~/go/src/github.com/hyperledger
git clone https://github.com/hyperledger/fabric.git
```

由于 Fabric 一直在更新，所有我们并不需要最新的源码，只需切换到 v1.0.0 版本的源码即可。

```
cd ~/go/src/github.com/hyperledger/fabric
git checkout v1.0.0
```

6. Fabric Docker 镜像的下载

这个其实很简单，因为我们已经设置了 Docker Hub 镜像地址，所以下载也会很快。官方文件也提供了批量下载的脚本，我们直接运行它。

```
cd ~/go/src/github.com/hyperledger/fabric/examples/e2e_cli/
source download-dockerimages.sh -c x86_64-1.0.0 -f x86_64-1.0.0
```

这样就可以下载所有需要的 Fabric Docker 镜像了。由于我们设置了国内的镜像，所以下载应该是比较快的。

下载完毕后，我们运行以下命令检查下载的镜像列表：

```
docker images
```

得到的结果如下：

```
REPOSITORY                    TAG                 IMAGE ID            CREATED             SIZE
hyperledger/fabric-tools      latest              0403fd1c72c7        6 weeks ago         1.32GB
hyperledger/fabric-tools      x86_64-1.0.0        0403fd1c72c7        6 weeks ago         1.32GB
hyperledger/fabric-couchdb    latest              2fbdbf3ab945        6 weeks ago         1.48GB
hyperledger/fabric-couchdb    x86_64-1.0.0        2fbdbf3ab945        6 weeks ago         1.48GB
hyperledger/fabric-kafka      latest              dbd3f94de4b5        6 weeks ago         1.3GB
hyperledger/fabric-kafka      x86_64-1.0.0        dbd3f94de4b5        6 weeks ago         1.3GB
hyperledger/fabric-zookeeper  latest              e545dbf1c6af        6 weeks ago         1.31GB
hyperledger/fabric-zookeeper  x86_64-1.0.0        e545dbf1c6af        6 weeks ago         1.31GB
hyperledger/fabric-orderer    latest              e317ca5638ba        6 weeks ago         179MB
hyperledger/fabric-orderer    x86_64-1.0.0        e317ca5638ba        6 weeks ago         179MB
hyperledger/fabric-peer       latest              6830dcd7b9b5        6 weeks ago         182MB
hyperledger/fabric-peer       x86_64-1.0.0        6830dcd7b9b5        6 weeks ago         182MB
hyperledger/fabric-javaenv    latest              8948126f0935        6 weeks ago         1.42GB
hyperledger/fabric-javaenv    x86_64-1.0.0        8948126f0935        6 weeks ago         1.42GB
hyperledger/fabric-ccenv      latest              7182c260a5ca        6 weeks ago         1.29GB
hyperledger/fabric-ccenv      x86_64-1.0.0        7182c260a5ca        6 weeks ago         1.29GB
```

7. 启动 Fabric

我们仍然停留在 e2e_cli 文件夹中，这里提供了启动、关闭 Fabric 网络的自动化脚本。我们要启动 Fabric 网络，并自动运行 Example02 ChainCode 的测试。执行以下命令：

```
./network_setup.sh up
```

这个命令进行了以下操作：

1）编译生成 Fabric 公私钥、证书的程序，程序在目录 fabric/release/linux-amd64/bin 下。

2）基于 configtx.yaml 生成创世区块和通道相关信息，并保存在 channel-artifacts 文件夹中。

3）基于 crypto-config.yaml 生成公私钥和证书信息，并保存在 crypto-config 文件夹中。

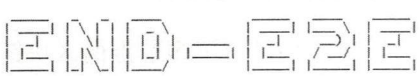
4）基于 docker-compose-cli.yaml 启动 1Orderer+4Peer+1CLI 的 Fabric 容器。

5）在 CLI 启动的时候，会运行 scripts/script.sh 文件，这个脚本文件包含了创建 Channel、加入 Channel、安装 Example02、运行 Example02 等功能。

最后运行完毕，我们可以看到这样的界面：

```
2018-05-07 09:48:57.516 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-07 09:48:57.517 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-07 09:48:57.517 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-07 09:48:57.517 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-07 09:48:57.518 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A91070A6708031A0C0889C2C0D70510...6D7963631A0A0A0571756572790A0161
2018-05-07 09:48:57.518 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 70C83B545648811F4E12DEFBCD62795F49650125C41B70D1B12517C68479C745
Query Result: 90
2018-05-07 09:49:20.174 UTC [main] main -> INFO 007 Exiting.....
===================== Query on PEER3 on channel 'mychannel' is successful =====================

===================== All GOOD, End-2-End execution completed =====================
```



如果看到这个界面，就说明整个 Fabric 网络已经通了。

8. 测试 Fabric 网络

下面仍然以现在安装好的 Example02 为例。在官方例子中，通道名是 mychannel，链码的名字是 mycc。我们首先进入 CLI，重新打开一个命令行窗口，输入以下命令：

```
docker exec -it cli bash
```

运行以下命令可以查询 a 账户的余额：

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

可以看到余额是 90。

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
2018-05-09 01:59:25.403 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-09 01:59:25.404 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-09 01:59:25.404 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-09 01:59:25.404 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-09 01:59:25.404 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A95070A6708031A0C08FDABC9D70510...6D7963631A0A0A0571756572790A0161
2018-05-09 01:59:25.405 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: F264CB872A908F604090B9BCFB3A91FB7AEEF2D8E085C33FE38C819E99499884
Query Result: 90
2018-05-09 01:59:26.032 UTC [main] main -> INFO 007 Exiting.....
```

然后，我们试一试把 a 账户的余额转给 b 账户 20 元，运行以下命令：

```
peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem -C mychannel -n mycc -c '{"Args":["invoke","a","b","20"]}'
```

运行结果为：

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile /opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem  -C mychannel -n mycc -c '{"Args":["invoke","a","b","20"]}'
2018-05-09 02:00:36.169 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-09 02:00:36.169 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-09 02:00:36.177 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-09 02:00:36.178 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-09 02:00:36.178 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6608031A0B08C4ACC9D70510...696E766F6B650A01610A01620A023230
2018-05-09 02:00:36.179 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 7AB64B8FD6489D122FC4234D3E7F30BCD6F424355C86C30264DE0158B9ACDB9A
2018-05-09 02:00:36.200 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A94070A6608031A0B08C4ACC9D70510...34FBD689A32B840714A219B8FCF981A6
2018-05-09 02:00:36.200 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: B3790BB73FCBDDDD2FB53644C7240ADE5A07E47333F808217684AE686A22197D
2018-05-09 02:00:36.206 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009 ESCC invoke result: version:1 response:<status:200 message:"OK" > payload:"\n \006\263\271\304\027\
242\225Zu\367Wt\301w\206\n\017\377\357\201\023}\332\t\036\206\316\025\t\000\333N\022Y\nE\022\024\n\004lscc\022\014\n\n\n\004mycc\022\002\010\003\022-\n\004mycc\022%\n\007\n\001a
\022\002\010\007\n\007\n\001b\022\002\010\007\032\007\n\001a\032\00270\032\010\n\001b\032\003230\032\003\010\310\001\"\013\022\004mycc\032\0031.0" endorsement:<endorser:"\n\007O
rg1MSP\022\200\006-----BEGIN -----\nMIICGjCCACCgAwIBAgIRAKFEifPedvNGpyCWH358vJMwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDVNhbiBG\ncmFuY2lzY
28xGTAXBgNVBAoTEG9yZzEuZXhhbXBsZS5jb20xHDAaBgNVBAMTE2Nh\nLm9yZzEuZXhhbXBsZS5jb20wHhcNMTgwNTA3MDgyOTIywhcNMjgwNTA0MDgyOTIy\nWjBbMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMB
QGA1UEBxMN\nU2FuIEZyYW5jaXNjbzEfMB0GA1UEAxMWcGVlcjAub3JnMS5leGFtcGxlLmNvbTBZ\nMBMGByqGSM49AgEGCCqGSM49AwEHA0IABLqfxeKoz7pB9HuUVBY1LFEok4E8ZJg7\nGqk2RqT9tnJMcJannpY6CkAUv/xkr419w
JbKaXeky/mtZx+BHZzJ5pujTTBLMA4G\nA1UdDwEB/wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1UdIwQkMCKAINMzHEVGrBLu\nEb1KlSf3nLT7Nhoe/LcVKMRVvHGSB62TMAoGCCqGSM49BAMCA0gAMEUCIQCNMzSC\nv8+CMEkFjc34
lGcRxlhI+NpCiswNF8R8rsfKOAIgAhBTDJo+yDRZ5enjU977hpWU\nPxJEFh4eUJ6l8sGIDks=\n-----END -----\n" signature:"0E\002!\000\336;\365\314\332\023\024\013\367R\337\3712\"\263\020\261bn4\
tf1y\233z\314+\362\264*\020\002 R\301<7\325\324t\023\305;\035\225\222\372\215\0164\353\326\211\243+\204\007\024\242\031\270\354\371\201\246" >
2018-05-09 02:00:36.206 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200
2018-05-09 02:00:36.207 UTC [main] main -> INFO 00b Exiting.....
```

现在转账完毕。我们试一试再查询一下 a 账户的余额，没问题的话，应该只剩下 70 了。实际情况如下：

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
2018-05-09 02:00:40.795 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-09 02:00:40.795 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-09 02:00:40.795 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-09 02:00:40.795 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-09 02:00:40.796 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A95070A6708031A0C08C8ACC9D70510...6D7963631A0A0A0571756572790A0161
2018-05-09 02:00:40.796 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 7CEBBC9893502D28E504ACED409CD7D05FF5457FB283F405AE0B17908ADD6DC7
Query Result: 70
2018-05-09 02:00:40.811 UTC [main] main -> INFO 007 Exiting.....
```

果然，一切正常。最后我们要关闭 Fabric 网络，首先需要运行 exit 命令退出 CLI 容器。关闭 Fabric 的命令与启动类似，命令如下：

```
cd ~/go/src/github.com/hyperledger/fabric/examples/e2e_cli
./network_setup.sh down
```

现在整个 Fabric 的环境已经测试完毕，一切正常。接下来我们就部署多节点的区块链网络和做自己的区块链开发了。

4.3.2 多节点区块链网络部署

1. 部署准备

我们要部署的是 4 peer+1 orderer 的架构，也就是官方的 e2c_cli 架构。为此我们需要准备 5 台机器。我们可以运行 5 台虚拟机，也可以购买 5 台云服务器，不管怎么样，需要这 5 台机器能够网络互通，而且安装相同的系统，都是 Ubuntu 16.04 版。为了方便，建议先启

用 1 台虚拟机，在其中把准备工作做完，然后基于这台虚拟机，再复制出 4 台即可。以下是用到的 5 台 Server 的主机名（角色）和 IP。

orderer.example.com	10.174.13.185
peer0.org1.example.com	10.51.120.220
peer1.org1.example.com	10.51.126.19
peer0.org2.example.com	10.51.116.133
peer1.org2.example.com	10.51.126.5

接下来，需要准备软件环境，包括 Go、Docker、Docker Compose，参考之前的章节即可。

2. docker-compose 配置文件准备

在 Fabric 的源码中，提供了单机部署 4Peer+1Orderer 的示例，存放在 Example/e2e_cli 文件夹中。我们可以在其中一台机器上运行单机的 Fabric 实例，确认无误后，在该机器上生成公私钥，修改该机器中的 Docker-compose 配置文件，然后把这些文件分发给另外 4 台机器。下面就以 orderer.example.com 这台机器为例进行部署。

（1）单机运行 4+1 Fabric 实例，确保脚本和镜像正常

我们先进入这个文件夹，然后直接运行以下命令：

```
./network_setup.sh up
```

这个命令可以在本机启动 4+1 的 Fabric 网络并且进行测试，运行 Example02 这个 ChainCode。我们可以看到每一步的操作，最后确认单机没有问题。确认镜像和脚本都是正常的，就可以关闭 Fabric 网络，继续多机 Fabric 网络设置工作。运行关闭 Fabric 命令：

```
./network_setup.sh down
```

（2）生成公私钥、证书、创世区块等

公私钥和证书用于 Server 和 Server 之间的安全通信，另外要创建 Channel 并让其他节点加入 Channel 就需要创世区块，这些必备文件都可以用一个命令生成，官方已经给出了命令脚本。

```
./generateArtifacts.sh mychannel
```

运行这个命令后，系统会创建 channel-artifacts 文件夹，里面包含了 mychannel 这个通道相关的文件。另外还有一个 crypto-config 文件夹，里面包含了各个节点的公私钥和证书的信息。

（3）设置 peer 节点的 docker-compose 文件

e2e_cli 中提供了多个 yaml 文件，可以基于 docker-compose-cli.yaml 文件创建。

```
cp docker-compose-cli.yaml docker-compose-peer.yaml
```

然后修改 docker-compose-peer.yaml，去掉 orderer 的配置，只保留一个 peer 和 cli。因

为我们要多级部署，节点与节点之间又是通过主机名通讯，所以需要修改容器中的 host 文件，也就是 extra_hosts 设置。修改后的 peer 配置如下：

```
    peer0.org1.example.com:
        container_name: peer0.org1.example.com
        extends:
            file:  base/docker-compose-base.yaml
            service: peer0.org1.example.com
        extra_hosts:
            - "orderer.example.com:10.174.13.185"
```

同样，cli 也需要能够与各个节点通讯，所以 cli 下面也需要添加 extra_hosts 设置，去掉无效的依赖，并且去掉 command 这一行。因为每个 peer 都会有个对应的客户端，也就是 cli，所以只需要手动执行一次命令，而不是自动运行。修改后的 cli 配置如下：

```
    cli:
        container_name: cli
        image: hyperledger/fabric-tools
        tty: true
        environment:
            - GOPATH=/opt/gopath
            - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
            - CORE_LOGGING_LEVEL=DEBUG
            - CORE_PEER_ID=cli
            - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
            - CORE_PEER_LOCALMSPID=Org1MSP
            - CORE_PEER_TLS_ENABLED=true
            - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/
server.crt
            - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/tls/
server.key
            - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer0.org1.example.
com/tls/ca.crt
            - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.com/msp
        working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
        volumes:
                - /var/run/:/host/var/run/
                - ../chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/
examples/chaincode/go
                - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/
                - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
                - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/
peer/channel-artifacts
        depends_on:
```

```
        - peer0.org1.example.com
    extra_hosts:
        - "orderer.example.com:10.174.13.185"
        - "peer0.org1.example.com:10.51.120.220"
        - "peer1.org1.example.com:10.51.126.19"
        - "peer0.org2.example.com:10.51.116.133"
        - "peer1.org2.example.com:10.51.126.5"
```

在单机模式下，4 个 peer 会映射主机不同的端口，但是在多机部署的时候是不需要映射不同端口的，所以需要修改 base/docker-compose-base.yaml 文件，将所有 peer 的端口映射都改为相同的。

```
ports:
    - 7051:7051
    - 7052:7052
    - 7053:7053
```

（4）设置 orderer 节点的 docker-compose 文件

与创建 peer 的配置文件类似，我们也复制一个 yaml 文件进行修改。

```
cp docker-compose-cli.yaml docker-compose-orderer.yaml
```

orderer 服务器上只需要保留 order 设置，其他 peer 和 cli 设置都可以删除。orderer 可以不设置 extra_hosts。

（5）分发配置文件

前面 4 步的操作都是在 orderer.example.com 上完成的。接下来需要将这些文件分发到另外 4 台服务器上。Linux 之间的文件传输可以使用 scp 命令。

先登录 peer0.org1.example.com，将本地的 e2e_cli 文件夹删除。

```
rm e2e_cli -R
```

然后再登录到 orderer 服务器上，退回到 examples 文件夹，因为这样可以方便地把其下的 e2e_cli 文件夹整个传到 peer0 服务器上。

```
scp -r e2e_cli fabric@10.51.120.220:/home/fabric/go/src/github.com/
hyperledger/fabric/examples/
```

我们在前面配置的就是 peer0.org1.example.com 上的节点，所以复制过来后不需要做任何修改即可。

再次运行 scp 命令，将配置文件复制到 peer1.org1.example.com 上，然后我们需要对 docker-compose-peer.yaml 做一个小小的修改，将启动的容器改为 peer1.org1.example.com，并且添加 peer0.org1.example.com 的 IP 映射，对应的 cli 中也改成对 peer1.org1.example.com 的依赖。修改后的 peer1.org1.example.com 上的配置文件如下：

```
version: '2'
```

```
    services:
        peer1.org1.example.com:
            container_name: peer1.org1.example.com
            extends:
                file:  base/docker-compose-base.yaml
                service: peer1.org1.example.com
            extra_hosts:
                - "orderer.example.com:10.174.13.185"
                - "peer0.org1.example.com:10.51.120.220"
        cli:
            container_name: cli
            image: hyperledger/fabric-tools
            tty: true
            environment:
                - GOPATH=/opt/gopath
                - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
                - CORE_LOGGING_LEVEL=DEBUG
                - CORE_PEER_ID=cli
                - CORE_PEER_ADDRESS=peer1.org1.example.com:7051
                - CORE_PEER_LOCALMSPID=Org1MSP
                - CORE_PEER_TLS_ENABLED=true
                - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.
com/tls/server.crt
                - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.
com/tls/server.key
                - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org1.example.com/peers/peer1.org1.example.
com/tls/ca.crt
                - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/hyperledger/
fabric/peer/crypto/peerOrganizations/org1.example.com/users/Admin@org1.example.
com/msp
            working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
            volumes:
                - /var/run/:/host/var/run/
                - ../chaincode/go/:/opt/gopath/src/github.com/hyperledger/fabric/
examples/chaincode/go
                - ./crypto-config:/opt/gopath/src/github.com/hyperledger/fabric/
peer/crypto/
                - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/peer/scripts/
                - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger/fabric/
peer/channel-artifacts
            depends_on:
                - peer1.org1.example.com
            extra_hosts:
                - "orderer.example.com:10.174.13.185"
                - "peer0.org1.example.com:10.51.120.220"
                - "peer1.org1.example.com:10.51.126.19"
                - "peer0.org2.example.com:10.51.116.133"
                - "peer1.org2.example.com:10.51.126.5"
```

接下来继续使用 scp 命令将 orderer 上的文件夹传送给 peer0.org2.example.com 和 peer1.org2.example.com，然后也要修改一下 docker-compose-peer.yaml 文件，使得其启动对应的 peer 节点。

3. 启动 Fabric 网络

现在所有文件都已经准备完毕，可以启动 Fabric 网络了。

（1）启动 orderer

首先启动 orderer 节点，在 orderer 服务器上运行以下命令：

```
docker-compose -f docker-compose-orderer.yaml up -d
```

运行完毕后我们可以使用 docker ps 看到运行了一个名字为 orderer.example.com 的节点。

```
root@orderer:~/go/src/github.com/hyperledger/fabric/examples/e2e_cli# docker-compose -f docker-compose-orderer.yaml up -d
Creating orderer.example.com
```

（2）启动 peer

然后切换到 peer0.org1.example.com 服务器，启动本服务器的 peer 节点和 cli。命令如下：

```
docker-compose -f docker-compose-peer.yaml up -d
```

运行完毕后使用 docker ps 应该可以看到两个正在运行的容器。

```
root@peer1:~/go/src/github.com/hyperledger/fabric/examples/e2e_cli# docker ps
CONTAINER ID        IMAGE                        COMMAND             CREATED             STATUS              PORTS                                NAMES
7e31e7ff7f40        hyperledger/fabric-tools     "/bin/bash"         5 minutes ago       Up 5 minutes                                             cli
fd3ad8ccb121        hyperledger/fabric-peer      "peer node start"   5 minutes ago       Up 5 minutes        0.0.0.0:7051-7053->7051-7053/tcp     peer1.org1.example.com
```

接下来依次在另外 3 台服务器运行启动 peer 节点容器的命令：

```
docker-compose -f docker-compose-peer.yaml up -d
```

```
root@peer1:~/go/src/github.com/hyperledger/fabric/examples/e2e_cli# docker-compose -f docker-compose-peer.yaml up -d
Creating network "e2ecli_default" with the default driver
Creating peer1.org1.example.com
Creating cli
```

现在，整个 Fabric 4+1 服务器网络已经成型。接下来创建 Channel 和运行 ChainCode。

（3）创建 Channel 测试 ChainCode

切换到 peer0.org1.example.com 服务器上，使用该服务器上的 cli 来运行创建 Channel 和运行 ChainCode 的操作。首先进入 cli 容器。

```
docker exec -it cli bash
```

进入容器后可以看到命令提示变为：

```
root@09d5af2a9820: /opt/gopath/src/github.com/hyperledger/fabric/peer
```

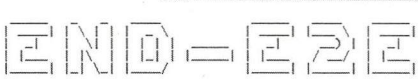
说明我们已经以 root 的身份进入 cli 容器内部。官方已经提供了完整的创建 Channel 和测试 ChainCode 的脚本，并且已经映射到 cli 容器内部，所以我们只需要在 cli 内运行如下命令：

```
./scripts/script.sh mychannel
```

那么该脚本就可以一步一步地自动完成创建 Channel，将其他节点加入 Channel，更新锚节点，创建 ChainCode，初始化账户，查询，转账，再次查询等链上代码的各个操作。直到最后，系统提示如下：

```
2018-05-07 09:48:57.516 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-07 09:48:57.517 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-07 09:48:57.517 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-07 09:48:57.517 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-07 09:48:57.518 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A91070A6708031A0C0889C2C0D70510...6D7963631A0A0A0571756572790A0161
2018-05-07 09:48:57.518 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 70C83B545648811F4E12DEFBCD62795F49650125C41B70D1812517C68479C745
Query Result: 90
2018-05-07 09:49:20.174 UTC [main] main -> INFO 007 Exiting.....
=================== Query on PEER3 on channel 'mychannel' is successful ===================

==================== All GOOD, End-2-End execution completed ====================
```



说明 Fabric4+1 多级部署成功了。现在是在 peer0.org1.example.com 的 cli 容器内，也可以切换到 peer0.org2.example.com 服务器，运行 docker ps 命令，可以看到本来是两个容器的，现在已经变成了 3 个容器，因为 ChainCode 会创建一个容器。

```
root@peer0:~# docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED             STATUS              PORTS                                NAMES
c45de4e0bb37        dev-peer0.org1.example.com-mycc-1.0    "chaincode -peer.a..."   16 hours ago        Up 16 hours                                              dev-peer0.org1.exam
ple.com-mycc-1.0
09d5af2a9820        hyperledger/fabric-tools               "/bin/bash"              16 hours ago        Up 16 hours                                              cli
b2b5a326533c        hyperledger/fabric-peer                "peer node start"        16 hours ago        Up 16 hours         0.0.0.0:7051-7053->7051-7053/tcp     peer0.org1.example.
com
```

（4）结果验证

查询 a 账户的金额。

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
```

结果如下：

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
2018-05-08 02:00:24.025 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-08 02:00:24.025 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-08 02:00:24.025 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-08 02:00:24.025 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-08 02:00:24.026 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6608031A0B08B889C4D70510...6D7963631A0A0A0571756572790A0161
2018-05-08 02:00:24.026 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 86C02CEC2FA7AD5CD8CA6881F50CC528B08A253232D8E847FFD1598A050109F2
Query Result: 30
2018-05-08 02:00:24.045 UTC [main] main -> INFO 007 Exiting.....
```

查询 b 账户的金额。

```
peer chaincode query -C mychannel -n mycc -c '{"Args":["query","b"]}'
```

结果如下：

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mycc -c '{"Args":["query","b"]}'
2018-05-08 02:02:20.581 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-08 02:02:20.581 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-08 02:02:20.582 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-08 02:02:20.582 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-08 02:02:20.582      [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A95070A6708031A0C08AC8AC4D70510...6D7963631A0A0A0571756572790A0162
2018-05-08 02:02:20.583 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 302E0EE6C81AD65DE5BA6BB9A227F51443DF5889730599398E4583EFF58D6B24
Query Result: 270
2018-05-08 02:02:20.599 UTC [main] main -> INFO 007 Exiting.....
```

转账 50 命令如下：

```
peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile /opt/
gopath/src/github.com/hyperledger/fabric/peer/crypto/ordererOrganizations/example.
com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem  -C
mychannel -n mycc -c '{"Args":["invoke","b","a","50"]}'
```

结果如下：

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile /opt/gopath/src/github.com/hyperledg
er/fabric/peer/crypto/ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/tlsca.example.com-cert.pem  -C mychannel -n mycc -c '{"Args":["invoke","b","a"
,"50"]}'
2018-05-08 02:02:44.034 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-08 02:02:44.034 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-08 02:02:44.042 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-08 02:02:44.042 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-08 02:02:44.043 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6608031A0B08C48AC4D70510...696E766F6B650A01620A01610A023530
2018-05-08 02:02:44.043 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: EFF205888BEEC0A65072512C76FB4A863C59B79BC41AF2421479620BB59ECCFB
2018-05-08 02:02:44.065 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A94070A6608031A0B08C48AC4D70510...2EE5843468EBF8D6AE75741EB6ED243F
2018-05-08 02:02:44.065 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: 85335C458E741F21D856A6FC945E104C97399C4048D0DE49C32908A2CC1B58AA
2018-05-08 02:02:44.070 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009 ESCC invoke result: version:1 response:<status:200 message:"OK" > payload:"\n \214\373E\274\217\210
,g\260\0170].\354\252Yy!0/\003\346 \252\264i\364\027\355\257D\322\022Y\nE\022\024\n\004lscc\022\014\n\n\n\004mycc\022\002\010\003\022-\n\004mycc\022%\n\007\n\001a\022\002\010\00
5\n\007\n\001b\022\002\010\005\032\007\n\001a\032\00280\032\010\n\001b\032\003220\032\003\010\310\001\"\013\022\004mycc\032\0031.0" endorsement:<endorser:"\n\007Org1MSP\022\200\
006-----BEGIN -----\nMIICGjCCACCgAwIBAgIRAKFEifPedvNGpyCWH358vJMwCgYIKoZIzj0EAwIwczEL\nMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBACTDVNhbiBG\ncmFuY2lzY28xGTAXBgNVBAoT
EG9yZzEuZXhhbXBsZS5jb20xHDAaBgNVBAMTE2Nh\nLm9yZzEuZXhhbXBsZS5jb20wHhcNMTgwNTA3MDgyOTIyWhcNMjgwNTA0MDgyOTIy\nWjBbMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMN\nU2F
uIEZyYW5jaXNjbzEfMB0GA1UEAxMWcGVlcjAub3JnMS5leGFtcGxlLmNvbTBZ\nMBMGByqGSM49AgEGCCqGSM49AwEHA0IABLqfxeKoz7pB9HuUVBY1LFEok4E8ZJg7\nGqk2RqT9tnJMcJannpY6CkAUv/xkr4l9wJbKaXeky/mtZx+B
HZzJ5pujTTBLMA4G\nA1UdDwEB/wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1UdIwQkMCKAINMzHEvGrBLu\nEb1KlSf3nLT7Nhoe/LCVKMRVvHGSB62TMAoGCCqGSM49BAMCA0gAMEUCIQCNMzSC\nv8+CMEkFjc34lGcRxlhI+NpCisw
NF8R8rsfKOAIgAhBTDJo+yDRZ5enjU977hpWU\nPxJEFh4eUJ6l8sGIDks=\n-----END -----\n" signature:"0D\002 p\344?\223C\345\332\000\211;\340\034,\306\001\255\241\220\253w\3226yR\017\370}\2
15\254=E\353\002 V\317\035L\364\275\217\367\231\3545e$\2400\310.\345\2044h\353\370\326\256ur\036\266\355$?"
2018-05-08 02:02:44.071 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200
2018-05-08 02:02:44.071 UTC [main] main -> INFO 00b Exiting.....
```

再查询一下 a、b 的账户金额，结果如下：

```
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mycc -c '{"Args":["query","a"]}'
2018-05-08 02:03:58.229 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-08 02:03:58.229 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-08 02:03:58.229 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-08 02:03:58.229 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-08 02:03:58.230 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6608031A0B088E8BC4D70510...6D7963631A0A0A0571756572790A0161
2018-05-08 02:03:58.230 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 8BD71EABF2ACC97B6F4CCF06664857CE158D6955DEF86F7CD38B62AD7D107E3C
Query Result: 80
2018-05-08 02:03:58.247 UTC [main] main -> INFO 007 Exiting.....
root@09d5af2a9820:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode query -C mychannel -n mycc -c '{"Args":["query","b"]}'
2018-05-08 02:04:03.071 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2018-05-08 02:04:03.072 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2018-05-08 02:04:03.072 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2018-05-08 02:04:03.072 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2018-05-08 02:04:03.073 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6608031A0B08938BC4D70510...6D7963631A0A0A0571756572790A0162
2018-05-08 02:04:03.073 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 5D9A66EB870FDD90CAE26ED456BFD2A4CDB68635194CEF52D314FFA527B730C1
Query Result: 220
2018-05-08 02:04:03.091 UTC [main] main -> INFO 007 Exiting.....
```

数据转账正常，说明 Fabric 多节点部署正常。

4. 总结

在 Fabric 多机部署的过程中还是有不少陷阱的，前后花了两天的时间才排除完毕，最终实现了 4+1 多机部署。其中与单机部署最大的不同的地方就是，在单机部署的时候，我们是在同一个 docker 网络中，所以相互之间通过主机名通讯很容易。而在多机环境中，就需要额外设置 DNS 或者通过 extra_hosts 参数设置容器中的 hosts 文件，而且不能一股脑地像 cli 一样把 5 台机器的域名 IP 配置到 peer 中，那样会报错的，所以只需要设置的需要的即可。

官方给的脚本已经替我们做了很多工作，同时也隐藏了很多细节，所以目前我们并不真正了解其内部的实现过程。在对 Fabric 源码进行解析后我们才能深入了解其细节。

4.4 Fabric 开发

4.4.1 ChainCode 开发

Fabric 的智能合约是通过链上代码 ChainCode 实现的，链上代码支持用 Java 或者 Go 语言进行开发。因为 Fabric 本身是用 Go 开发的，所以本节用 Go 进行 ChainCode 的开发讲解。

ChainCode 的 Go 代码需要定义 SimpleChaincode 这样一个 struct，然后在该 struct 上定义 Init 和 Invoke 两个函数。然后还要定义一个 main 函数，作为 ChainCode 的启动入口。以下是 ChainCode 的模板：

```
package main
import (
"github.com/hyperledger/fabric/core/chaincode/shim"
    pb "github.com/hyperledger/fabric/protos/peer"
```

```
    "fmt"
    )

    type SimpleChaincode struct {
    }

    func main() {
        err := shim.Start(new(SimpleChaincode))
        if err != nil {
            fmt.Printf("Error starting Simple chaincode: %s", err)
        }
    }
    func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        return shim.Success(nil)
    }


    func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
        function, args := stub.GetFunctionAndParameters()
        fmt.Println("invoke is running " + function)
        if function == "test1" {// 自定义函数名称
            return t.test1(stub, args)// 定义调用的函数     }
        return shim.Error("Received unknown function invocation")
    }
    func (t *SimpleChaincode) test1(stub shim.ChaincodeStubInterface, args []
string) pb.Response{
        return shim.Success([]byte("Called test1"))
    }
```

从以上可以看到，在调用 Init 和 Invoke 的时候，都会传入参数 stub shim.ChaincodeStub-Interface，这个参数提供的接口为我们编写 ChainCode 的业务逻辑提供了大量实用的方法。

1. 获得调用的参数

前面给出的 ChainCode 的模板中，我们已经可以看到，在调用 Invoke 的时候，传入的参数决定我们具体调用了哪个方法，所以需要先使用 GetFunctionAndParameters 解析调用时传入的参数。除了这个方法以外，接口还提供了另外几个方法，不过其本质都是一样的。

- GetArgs()[][]byte 以 byte 数组的形式获得传入的参数列表。
- GetStringArgs()[]string 以字符串数组的形式获得传入的参数列表。
- GetFunctionAndParameters()(string, []string) 将字符串数组的参数分为两部分，数组第 1 个字是 Function，剩下的都是 Parameter。

GetArgsSlice()([]byte, error) 以 byte 切片的形式获得参数列表。

2. 增删改查 State DB

对于 ChainCode 来说，核心的操作就是对 State Database 的增删改查。对此，Fabric 接口提供了 3 个对 State DB 的操作方法。

（1）增改数据

```
PutState(key string, value []byte) error
```

对于 State DB 来说，增加和修改数据是统一的操作，因为 State DB 是一个 Key Value 数据库，如果我们指定的 Key 在数据库中已经存在，那么就是修改操作；如果 Key 不存在，那么就是插入操作。对于实际的系统来说，我们的 Key 可能是单据编号，或者是系统分配的自增 ID+ 实体类型作为前缀，而 Value 则是一个对象经过 Json 序列号后的字符串。比如，我们定义一个 Student 的 Struct，然后插入一个学生数据，对应的代码如下：

```
type Student struct {
    Id int
    Name string
}
func (t *SimpleChaincode) testStateOp(stub shim.ChaincodeStubInterface, args []
string) pb.Response{
    student1:=Student{1,"Devin Zeng"}
    key:="Student:"+strconv.Itoa(student1.Id)          //Key 格式为 Student:{Id}
    studentJsonBytes, err := json.Marshal(student1)    //Json 序列号
    if err != nil {
        return shim.Error(err.Error())
    }
    err= stub.PutState(key,studentJsonBytes)
    if(err!=nil){
        return shim.Error(err.Error())
    }
    return shim.Success([]byte("Saved Student!"))
}
```

（2）删除数据

```
DelState(key string) error
```

这个也很好理解，根据 Key 删除 State DB 的数据。如果根据 Key 找不到对应的数据，则删除失败。

```
err= stub.DelState(key)if err != nil {
    return shim.Error("Failed to delete Student from DB, key is: "+key)
}
```

（3）查询数据

```
GetState(key string) ([]byte, error)
```

因为我们是 Key Value 数据库，所以根据 Key 来对数据库进行查询是一件很常见。很高效的操作。返回的数据是 byte 数组，需要转换为 string，然后再 Json 反序列化，才可以得到我们想要的对象。

```
dbStudentBytes,err:= stub.GetState(key)var dbStudent Student;
```

```
    err=json.Unmarshal(dbStudentBytes,&dbStudent)//反序列化 if err != nil {
        return shim.Error("{\"Error\":\"Failed to decode JSON of: " + string(dbStudentBytes)+
"\" to Student}")
    }
    fmt.Println("Read Student from DB, name:"+dbStudent.Name)
```

不能在一个 ChinCode 函数中 PutState 后又马上 GetState，这个时候 GetState 是没有最新值的，因为这时 Transaction 并没有完成，还没有提交到 StateDB 里面。

3. 复合键的处理

（1）生成复合键

```
CreateCompositeKey(objectType string, attributes []string) (string, error)
```

前面，在进行数据库的增删改查的时候，都需要用到 Key，而我们使用的是自己定义的 Key 格式：{StructName}:{Id}。若只有单主键 Id 还比较简单，如果有多个列做联合主键怎么办？实际上，ChainCode 也提供了生成 Key 的方法 CreateCompositeKey，通过这个方法，我们可以将联合主键涉及的属性都传进去，只需声明对象的类型即可。

以选课表为例，里面包含了以下属性：

```
type ChooseCourse struct {
    CourseNumber string  //开课编号
    StudentId int        //学生 ID
    Confirm bool         //是否确认
}
```

其中 CourseNumber+StudentId 构成了这个对象的联合主键，我们要获得生成的复核主键，那么可写为：

```
    cc:=ChooseCourse{"CS101",123,true}  var key1,_= stub.CreateCompositeKey("Choos
eCourse",[]string{cc.CourseNumber,strconv.Itoa(cc.StudentId)})
    fmt.Println(key1)
```

其实 Fabric 是用 U+0000 来把各个字段分割开的，因为这个字符太特殊，所以很适合做分割符号

（2）拆分复合键

```
SplitCompositeKey(compositeKey string) (string, []string, error)
```

既然有组合那么就有拆分。当我们从数据库中获得了一个复合键的 Key 之后，怎么知道其具体是由哪些字段组成的呢？其实就是用 U+0000 把这个复合键再分割开，得到结果中第一个是 objectType，剩下的就是复合键用到的列的值。

```
objType,attrArray,_:= stub.SplitCompositeKey(key1)
fmt.Println("Object:"+objType+" ,Attributes:"+strings.Join(attrArray,"|"))
```

（3）部分复合键的查询

```
GetStateByPartialCompositeKey(objectType string, keys []string)
(StateQueryIteratorInterface, error)
```

这里其实是一种对 Key 进行前缀匹配的查询，也就是说，虽然是部分复合键的查询，但是不允许用后面部分的复合键进行匹配，必须是前面部分。

4. 获得当前用户

获得当前用户方法如下：

```
GetCreator() ([]byte, error)
```

这个方法可以获得调用这个 ChainCode 的客户端的用户的证书，这里虽然返回的是 byte 数组，但是其实是一个字符串，内容格式如下：

```
-----BEGIN CERTIFICATE-----
MIICGjCCAcCgAwIBAgIRAMVe0+QZL+67Q+R2RmqsD90wCgYIKoZIzj0EAwIwczEL
MAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcTDVNhbiBG
cmFuY2lzY28xGTAXBgNVBAoTEG9yZzEuZXhhbXBsZS5jb20xHDAaBgNVBAMTE2Nh
Lm9yZzEuZXhhbXBsZS5jb20wHhcNMTcwODEyMTYyNTU1WhcNMjcwODEwMTYyNTU1
WjBbMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMN
U2FuIEZyYW5jaXXXbzEfMB0GA1UEAwwWVXNlcjFAb3JnMS5leGFtcGxlLmNvbTBZ
MBMGByqGSM49AgEGCCqGSM49AwEHA0IABN7WqfFwWWKynl9SI87byp0SZO6QU1hT
JRatYysXX5MJJRzvvVsSTsUzQh5jmgwkPbFcvk/x4W8lj5d2Tohff+WjTTBLMA4G
A1UdDwEB/wQEAwIHgDAMBgNVHRMBAf8EAjAAMCsGA1UdIwQkMCKAIO2os1zK9BKe
Lb4P8lZOFU+3c0S5+jHnEILFWx2gNoLkMAoGCCqGSM49BAMCA0gAMEUCIQDAIDHK
gPZsgZjzNTkJgglZ7VgJLVFOuHgKWT9GbzhwBgIgE2YWoDpG0HuhB66UzlA+6QzJ
+jvM0tOVZuWyUIVmwBM=
-----END CERTIFICATE-----
```

我们常见的需求是在 ChainCode 中获得当前用户的信息，以便进行权限管理。那么我们怎么获得当前用户呢？我们可以把这个证书的字符串转换为 Certificate 对象。一旦转换成这个对象，我们就可以通过 Subject 获得当前用户的名字。

```
func (t *SimpleChaincode) testCertificate(stub shim.ChaincodeStubInterface,
args []string) pb.Response{
    creatorByte,_:= stub.GetCreator()
        certStart := bytes.IndexAny(creatorByte, "-----BEGIN")
        if certStart == -1 {
            fmt.Errorf("No certificate found")
        }
        certText := creatorByte[certStart:]
        bl, _ := pem.Decode(certText)
        if bl == nil {
            fmt.Errorf("Could not decode the PEM structure")
        }

        cert, err := x509.ParseCertificate(bl.Bytes)
```

```
    if err != nil {
        fmt.Errorf("ParseCertificate failed")
    }
    uname:=cert.Subject.CommonName
    fmt.Println("Name:"+uname)
    return shim.Success([]byte("Called testCertificate "+uname))
}
```

5. 高级查询

前面提到的 GetState 只是最基本的根据 Key 查询值的操作，但是在很多时候，我们需要查询返回的是一个集合，比如想要知道某个区间的 Key 对应的所有对象，或者我们需要查询 Value 对象内部的属性。

（1）Key 区间查询

这个方法提供了对某个区间的 Key 进行查询的接口，适用于任何 State DB。由于返回的是一个 StateQueryIteratorInterface 接口，我们需要通过这个接口再做一个 for 循环，才能读取返回的信息。所以可以独立出一个方法，专门将该接口返回的数据以 string 的 byte 数组形式返回。转换方法如下：

```
func getListResult(resultsIterator shim.StateQueryIteratorInterface) ([]
byte,error){

    defer resultsIterator.Close()
//buffer 是一个包含查询记录的 JSON 数组
    var buffer bytes.Buffer
    buffer.WriteString("[")

    bArrayMemberAlreadyWritten := false
    for resultsIterator.HasNext() {
        queryResponse, err := resultsIterator.Next()
        if err != nil {
            return nil, err
        }
        if bArrayMemberAlreadyWritten == true {
            buffer.WriteString(",")
        }
        buffer.WriteString("{\"Key\":")
        buffer.WriteString("\"")
        buffer.WriteString(queryResponse.Key)
        buffer.WriteString("\"")

        buffer.WriteString(", \"Record\":")
        buffer.WriteString(string(queryResponse.Value))
        buffer.WriteString("}")
        bArrayMemberAlreadyWritten = true
    }
    buffer.WriteString("]")
    fmt.Printf("queryResult:\n%s\n", buffer.String())
```

```
        return buffer.Bytes(), nil
    }
```

比如要查询编号从 1 号到 3 号的所有学生，那么查询代码可以这么写：

```
    func (t *SimpleChaincode) testRangeQuery(stub shim.ChaincodeStubInterface,
args []string) pb.Response{
        resultsIterator,err:= stub.GetStateByRange("Student:1","Student:3")
        if err!=nil{
            return shim.Error("Query by Range failed")
        }
        students,err:=getListResult(resultsIterator)
        if err!=nil{
            return shim.Error("getListResult failed")
        }
        return shim.Success(students)
    }
```

（2）富查询

富查询是对 Value 的内容进行查询，如果是 LevelDB，那么是不支持，只有 CouchDB 时才能用这个方法。传入的 query 这个字符串其实是 CouchDB 所使用的 Mango 查询，可以在其官方博客了解到一些信息：https://blog.CouchDB.org/2016/08/03/feature-mango-query/。其基本语法可以在 https://github.com/cloudant/mango 这里看到。

比如，仍然以前面的 Student 为例，我们要按 Name 来进行查询，那么代码可以写为：

```
    func (t *SimpleChaincode) testRichQuery(stub shim.ChaincodeStubInterface, args
[]string) pb.Response{
        name:="Devin Zeng"//这里按说应该是参数传入
        queryString := fmt.Sprintf("{\"selector\":{\"Name\":\"%s\"}}", name)
        resultsIterator,err:= stub.GetQueryResult(queryString)// 必须是 CouchDB 才行
        if err!=nil{
            return shim.Error("Rich query failed")
        }
        students,err:=getListResult(resultsIterator)
        if err!=nil{
            return shim.Error("Rich query failed")
        }
        return shim.Success(students)
    }
```

（3）历史数据查询

对同一个数据（也就是 Key 相同）的更改，会记录到区块链中，可以通过 GetHistoryForKey 方法获得这个对象在区块链中记录的更改历史，包括是在哪个 TxId 修改的数据或修改的时间戳，以及是否是删除等。比如之前的 Student:1 这个对象，我们更改和删除过数据，现在要查询这个对象的更改记录，那么对应代码为：

```
    func (t *SimpleChaincode) testHistoryQuery(stub shim.ChaincodeStubInterface,
```

```
args []string) pb.Response{
        student1:=Student{1,"Devin Zeng"}
        key:="Student:"+strconv.Itoa(student1.Id)
        it,err:= stub.GetHistoryForKey(key)
        if err!=nil{
            return shim.Error(err.Error())
        }
        var result,_= getHistoryListResult(it)
        return shim.Success(result)
    }
    func getHistoryListResult(resultsIterator shim.HistoryQueryIteratorInterface)
([]byte,error){

        defer resultsIterator.Close()
        //buffer 是一个包含查询记录的 JSON 数组
        var buffer bytes.Buffer
        buffer.WriteString("[")

        bArrayMemberAlreadyWritten := false
        for resultsIterator.HasNext() {
            queryResponse, err := resultsIterator.Next()
            if err != nil {
                return nil, err
            }
            if bArrayMemberAlreadyWritten == true {
                buffer.WriteString(",")
            }
            item,_:= json.Marshal( queryResponse)
            buffer.Write(item)
            bArrayMemberAlreadyWritten = true
        }
        buffer.WriteString("]")
        fmt.Printf("queryResult:\n%s\n", buffer.String())
        return buffer.Bytes(), nil
    }
```

6. 调用链上代码

调用另外的链上代码方法如下：

```
    InvokeChaincode(chaincodeName string, args [][]byte, channel string)
pb.Response
```

这个比较好理解，就是在链上代码中调用别人已经部署好的链上代码。比如，官方提供的 example02，我们要在代码中实现从 a 向 b 的转账，那么代码应该如下：

```
    func (t *SimpleChaincode) testInvokeChainCode(stub shim.ChaincodeStubInterface,
args []string) pb.Response{
        trans:=[][]byte{[]byte("invoke"),[]byte("a"),[]byte("b"),[]byte("11")}
        response:= stub.InvokeChaincode("mycc",trans,"mychannel")
        fmt.Println(response.Message)
```

```
    return shim.Success([]byte( response.Message))
}
```

这里需要注意，我们使用的是 example02 的链上代码的实例名 mycc，而不是代码名 example02。

7. 获得提案对象属性

（1）获得签名的提案

```
GetSignedProposal() (*pb.SignedProposal, error)
```

从客户端发现背书节点的 Transaction 或者 Query 都是一个提案，GetSignedProposal 获得当前的提案对象，包括客户端对这个提案的签名。

（2）获得 Transient 对象

```
GetTransient() (map[string][]byte, error)
```

Transient 是在提案中 Payload 对象中的一个属性，也就是 ChaincodeProposalPayload.TransientMap。

（3）获得交易时间戳

```
GetTxTimestamp() (*timestamp.Timestamp, error)
```

交易时间戳也是在提案对象中获取的，提案对象的 Header 部分也就是 proposal.Header.ChannelHeader.Timestamp。

（4）获得 Binding 对象

```
GetBinding() ([]byte, error)
```

这个 Binding 对象也是从提案对象中提取并组合出来的，其中包含 proposal.Header 中的 SignatureHeader.Nonce、SignatureHeader.Creator 和 ChannelHeader.Epoch。

8. 事件设置

事件设置方法如下：

```
SetEvent(name string, payload []byte) error
```

当 ChainCode 提交完毕后，会通过 Event 的方式通知 Client。而通知的内容可以通过 SetEvent 设置。

```
    func (t *SimpleChaincode) testEvent(stub shim.ChaincodeStubInterface, args []
string) pb.Response{
        tosend := "Event send data is here!"
        err := stub.SetEvent("evtsender", []byte(tosend))
        if err != nil {
            return shim.Error(err.Error())
        }
```

```
    return shim.Success(nil)
}
```

事件设置完毕后，需要在客户端也做相应的修改。

9. ChainCode API

作为 ChainCode 中的利器，shim.ChaincodeSubInterface 提供了一系列 API，供开发者在编写链码时灵活选择使用。

这些 API 可分为 4 类：账本状态交互 API、交易信息相关 API、参数读取 API、其他 API。下面分别介绍。

（1）账本状态交互 API

ChainCode 需要将一些数据记录在分布式账本中。需要记录的数据称为状态（state），以键值对（key-value）的形式存储。账本状态 API 可以对账本状态进行操作，这一点十分重要。方法的调用会更新交易提案的读写集合，在 Committer 进行验证时会在此执行。与账本状态进行比对，这类 API 的大致功能如下：

API	方法格式	说明
GetState	GetState(key string)([]byte.error)	负责查询账本，返回指定键对应的值
PutState	PutState(key string，value[]byte) error	尝试在账本中添加或更新一对键值。这一对键值会被添加到写集合中，等待 Committer 进一步的验证，验证通过后会真正写入账本
DelState	DelState(key string)error	在账本中删除一对键值。同样，将对键值的删除记录到交易提案的写集合中，等待 Committer 进一步的验证，验证通过后会真正写入账本
GetStateByRange	GetStateByRange(startKey，endKey string)(StateQueryIterator-Interface，error)	查询指定范围内的键值，startKey、endKey 分别指定起始（包括）和终止（不包括），当为空时默认是最大范围。返回结果是一个迭代器 StateQueryIterator-Interface 结构，可以按照字典序迭代每个键值对，最后需调用 Close() 方法关闭
GetStateByPartial CompositeKey	GetStateByParialCompositeKey (objectType string，keys []string) (StateQueryIteratorInterface，error)	根据局部的复合键（前缀）返回所有匹配的键值。返回结果也是一个迭代器 StateQueryIteratorInterface 结构，可以按照字典序迭代每个键值对，最后需调用 Close() 方法关闭
GetHistoryForKey	GetHistoryForKey(key string) (History QueryIteratorInterface，error)	返回某个键的所有历史值。需要在节点配置中打开历史数据库特性（ledger.history.enableHistoryDatabase = true）
GetQueryResult	GetQueryResult(query string)(State-QueryIteratorInterface,error)	对（支持富查询功能的）状态数据库进行富查询（richquery）。返回结果为迭代器结构 StateQueryIterator-Interface。注意该方法不会被 Committer 重新执行进行验证，因此，不应该用于更新账本状态的交易中。目前仅有 CouchDB 类型的状态数据库支持富查询

（2）交易信息相关 API

交易信息相关 API 可以获取到与交易信息自身相关的数据。用户对链码的调用（初始化和升级时调用 Init() 方法，运行时调用 Invoke() 方法）过程中会产生交易提案。这些 API 支持查询当前交易提案的一些属性，具体信息如下：

API	方法格式	说明
GetTxID	GetTxID()string。	该方法返回交易提案中指定的交易 ID。一般情况下，交易 ID 是在客户端生成提案时候产生的数字摘要，由 Nonce 随机串和签名者身份信息。一起进行 SHA256 哈希运行生成
GetTxTimestamp	GetTxTimestamp()(*timestamp.Timestamp，error)	返回交易被创建时的客户端打上的时间截。这个时间截是直接从交易 ChannelHeader 中提取的，所以在所有背书节点（endorsers）处看到的值都相同
GetBinding	GetBinding()([]byte，error)	返回交易的 binding 信息。 注意：交易的 binding 信息是将交易提案的 nonse、Creator、epoch 等信息组合起来，再进行哈希得到的数字摘要
GetSignedProposal	GetSignedProposal()(*pb.SignedProposal，error)	返回该 stub 的 SignedProposal 结构。包括与交易提案相关的所有数据
GetCreator	GetCreator()([]byte，error)	返回该交易的提交者的身份信息，从 signedProposal 中的 SignatoseHeader.Creator 提取
GetTransient	GetTransient()(map[string][]byte.error)	返回交易中带有的一些临时信息，从 ChaincodeProposal Payload transient 域提取，可以存放一些应用相关的保密信息，这些信息不会被写到账本中

（3）参数读取 API

调用链码时支持传入若干参数，参数可通过 API 读取。具体信息如下：

API	方法格式	说明
GetArgs	GetArgs()[][]byte	提取调用链码时交易 Proposal 中指定的参数，以字节串（ByteArray）数组形式返回。可以在 Init 或 Invoke 方法中使用。这些参数从 ChaincodeSpec 结构中的 Input 域直接提取
GetArgsSlice	GetArgslice()([]byte，error)	提取调用链码时交易 Proposal 中指定的参数，以字节串形式返回
GetFunctionAndParameters	GetFunctionAndParameters()(string，[]string)	提取调用链码时交易 Proposal 中指定的参数，其中第一个参数作为被调用的函数名称。剩下的参数作为函数的执行参数。这是链码开发者和用户约定俗成的习惯，即在 Init/Invoke 方法中编写实现若干子函数，用户调用时以第一个参数作为函数名，链码中的代码根据函数名称可以仅执行对应的分支处理逻辑
GetStringArgs	GetStringArgs()[]string	提取调用链码时交易 Proposal 中指定的参数，以字符串（String）数组形式返回

（4）其他 API

除了上面一些 API 以外还有一些辅助 API。具体如下：

API	方法格式	说明
CreateCompositeKey	CreateCompositeKey(objectType string，attributes []string)(string，error)	给定一组属性（attributes），该 API 将这些属性组合起来构造返回一个复合键。返回的复合键可以被 PutState 等方法使用。objectType 和 attributes 只允许合法的 utf8 字符串，并且不能包含 U+0000 和 U+10FFFF
SplitCompositeKey	SplitCompositeKey(compositeKey string)(string，[]string，error)	该方法与 CreateCompositeKey 方法相对，给定一个复合键，将其拆分为构造复合键时所用的属性
InvokeChaincode	InvokeChaincode(chaincodeName string，args[][]byte，channel string) pb.Response	调用另一个链码中的 Invoke 方法，如果被调用链码在同一个通道内，则添加其读写集合信息到调用交易；否则执行调用但不影响读写集合信息。如果 channel 为空，则默认为当前通道。目前仅限于读操作，同时不会生成新的交易
SetEvent	SetEvent(name string，payload []byte)error	设定当这个交易在 Committer 处被认证通过，写入区块时发送的事件（event）

10. 导入第三方包

在企业级应用开发中，经常会涉及流程和状态，而有限状态机（FSM）则是对应的一种简单实现，如果复杂化，就上升到 Workflow 和 BPM 了。我们在 Fabric ChainCode 的开发过程中，也有可能涉及状态机，这里我们就举一个例子，用 FSM 实现一个二级审批报销单的状态转移。

我们有一个报销单，员工填写报销单时可以保存为 Draft 状态，提交后变成 Submitted 状态。然后在一级审批的时候，可以同意（Approve）或者拒绝（Reject），同意了改为 L1Approved，进入下一级审批；拒绝了就以 Reject 状态打回给填表人。二级审批人员也是有 Approve 和 Reject 两个操作，同意了状态就改为 Complete；拒绝了就改为 Reject。这是一个很常见的审批例子，如图 4-5 所示。

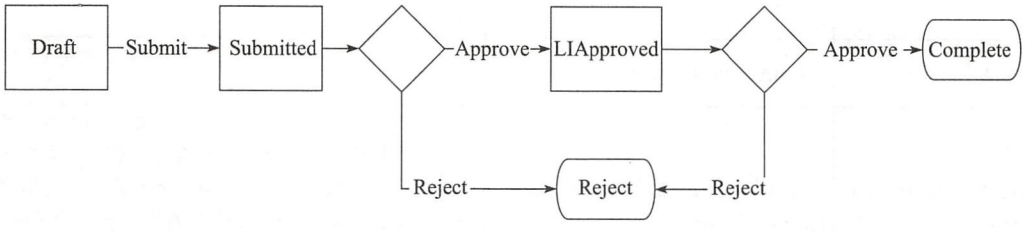


图 4-5 表单状态机示例图

我们使用 Go 来开发 ChainCode，那么可以采用 https://github.com/looplab/fsm 这个 FSM 库。这个库也是 Fabric 官方采用的状态机库。操作过程如下。

（1）新建 ChainCode 项目并引入 FSM 库

我们新建一个项目 fsmtest，并在其中建立 ChainCode 文件：main.go。然后新建 vendor 文件夹，将 https://github.com/looplab/fsm 从 GitHub 复制下来，并放在 vendor/github.com/looplab/fsm 文件夹中。最终项目的文件结构如图 4-6 所示。

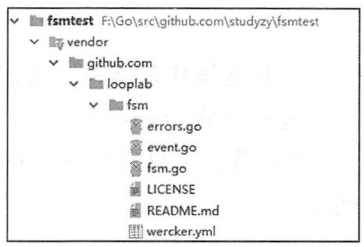


图 4-6　最终项目的文件结构

（2）定义 FSM 初始化函数

接下来打开 main.go 文件，除了编写 ChainCode 所必须使用的函数外，最主要的就是编写定义状态机转移的初始化函数了。根据前面流程图中的流程状态定义，可以写出如下的 FSM 初始化函数：

```
func InitFSM(initStatus string) *fsm.FSM{
    f := fsm.NewFSM(
        initStatus,
        fsm.Events{
            {Name: "Submit", Src: []string{"Draft"}, Dst: "Submited"},
            {Name: "Approve", Src: []string{"Submited"}, Dst: "L1Approved"},
            {Name: "Reject", Src: []string{"Submited"}, Dst: "Reject"},
            {Name: "Approve", Src: []string{"L1Approved"}, Dst: "Complete"},
            {Name: "Reject", Src: []string{"L1Approved"}, Dst: "Reject"},
        },
        fsm.Callbacks{},
    )
    return f;
}
```

（3）在 ChainCode 中调用 FSM Event

接下来我们在 ChainCode 重定义了 4 个函数：Draft、Submit、Approve、Reject。

于是我们可以在 Invoke 函数中定义 4 种情况。

```
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
    function, args := stub.GetFunctionAndParameters()
    fmt.Println("invoke is running " + function)
    if function == "Draft" {                    //自定义函数名称
        return t.Draft(stub, args)              //定义调用的函数
    } else if function == "Submit" {
        return FsmEvent(stub,args,"Submit")
    }  else if function == "Approve" {
        return FsmEvent(stub,args,"Approve")
    }  else if function == "Reject" {
        return FsmEvent(stub,args,"Reject")
    }
    return shim.Error("Received unknown function invocation")
}
```

其中 Draft 函数是把表单状态初始化为 Draft 并保存到数据库，并不涉及状态的修改。

```
func (t *SimpleChaincode) Draft(stub shim.ChaincodeStubInterface, args []
```

```
string) pb.Response{
        formNumber:=args[0]
        status:="Draft"
        stub.PutState(formNumber,[]byte(status))//初始化 Draft 状态的表单保存到 StateDB
        return shim.Success([]byte(status))
    }
```

而其他操作都涉及状态的修改。由于我们引入了状态机，所以只需要初始化状态机，并发送对应的 Event 即可，而最新的状态是由状态机根据我们的定义而获得的。所以虽然有 3 个操作，却只需要一个函数就能完成，并没有冗余的 if else 判断，这就是使用状态机的优势。

```
    func FsmEvent(stub shim.ChaincodeStubInterface, args []string,event string)
pb.Response{
        formNumber:=args[0]
        bstatus,err:=stub.GetState(formNumber)  //从 StateDB 中读取对应表单的状态
        if err!=nil{
            return shim.Error("Query form status fail, form number:"+formNumber)
        }
        status:=string(bstatus)
        fmt.Println("Form["+formNumber+"] status:"+status)
        f:=InitFSM(status)                          //初始化状态机，并设置当前状态为表单的状态
        err=f.Event(event)                          //触发状态机的事件
        if err!=nil{
            return shim.Error("Current status is "+status+" does not support envent:"+event)
        }
        status=f.Current()
        fmt.Println("New status:"+status)
        stub.PutState(formNumber,[]byte(status))//更新表单的状态
        return shim.Success([]byte(status));    //返回新状态
    }
```

（4）部署并测试 ChainCode

现在状态写完了，我们需要进行测试，我们可以 git push 到 GitHub，然后到 Ubuntu 中 git clone 下来。也可以通过 rz 命令，把 Windows 中开发好的 ChainCode 上传到 Ubuntu 中。不管什么方法。最终我们整个 ChainCode 项目放在了 ~/go/src/github.com/hyperledger/fabric/examples/chaincode/go/fsmtest 这个文件夹中。

然后使用 e2e_cli 下面的 network_setup.sh up 命令启动整个 Fabric 网络。启动 Fabric 网络后，我们需要进入 CLI 进行部署和测试 fsmtest。

```
    docker exec -it cli bash
```

然后安装并初始化 ChainCode。

```
    peer chaincode install -n fsmtest -v 1.0 -p github.com/hyperledger/fabric/
examples/chaincode/go/fsmtest
    ORDERER_CA=/opt/gopath/src/github.com/hyperledger/fabric/peer/crypto/
```

```
ordererOrganizations/example.com/orderers/orderer.example.com/msp/tlscacerts/
tlsca.example.com-cert.pem
    peer chaincode instantiate -o orderer.example.com:7050 --tls true --cafile
$ORDERER_CA -C mychannel -n fsmtest -v 1.0 -c '{"Args":[]}'
```

安装完毕后，我们可以起草一个报销单 EXP1。

```
    peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile
$ORDERER_CA -C mychannel -n fsmtest -c '{"Args":["Draft","EXP1"]}'
```

我们可以看到系统返回的结果如下：

```
root@3711df2a20c4:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile $ORDERER_CA -C mychannel -n f
smtest -c '{"Args":["Draft","EXP1"]}'
2017-09-07 15:39:42.065 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-09-07 15:39:42.065 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-09-07 15:39:42.077 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-09-07 15:39:42.077 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-09-07 15:39:42.078 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A93070A6908031A0B088ED0C5CD0510...741A0D0A0544726166740A0445585031
2017-09-07 15:39:42.078 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: D043B4D95566281CD1E5CDE40F3CAD1E8478EC73BD897781F3F24E48A7B9376D
2017-09-07 15:40:08.520 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A93070A6908031A0B088ED0C5CD0510...675A412B333E24AC83A9FB25F93B5476
2017-09-07 15:40:08.522 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: BA93C74BF42603E61CB9332A3C018F5ED54473952B64A05FCA1507C1FAB00579
2017-09-07 15:40:08.550 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009 ESCC invoke result: version:1 response:<status:200 message:"OK" payload:"Draft" > payload:"\
n \345\214\244Ag\216V\362"et$m\260\356^\361|\002U{\226Y8:i\222\245\215A\032\022S\n5\022\032\n\007fsmtest\022\017\032\r\n\004EXP1\032\005Draft\022\027\n\004Isco\022\017\n
\r\n\007fsmtest\022\002\010\004\032\n\010\310\001\032\005Draft\"\016\022\007fsmtest\032\0031.0" endorsement:<endorser:"\n\007Org1MSP\022\374\005-----BEGIN -----\nMIICGDCC
Ab+gAwIBAgIQIf2jXWyVCUdAc7cHqjDxTDAKBggqhkjOPQQDAjBzMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNU2FuIEZy\nYW5jaXNjbzEZMBcGA1UEChMQb3JnMS5leGFtcGxlLmNvbT
EoMBoGA1UEAxMTY2Eu\nb3JnMS5leGFtcGxlLmNvbTAeFw0xNzA5MDcxNTM3NTJaFw0yNzA5MDUxNTM3NTJa\nMFsxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1T\nYW4gRnJhbmNpc2NvMR
8wHQYDVQQDExZwZWYyMC5vcmxLmV4YW1wbGUuY29tMFkw\nEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEY+g+r9cO+3m15XpMq7MkfS6GGMunWspB\n9g6f229RFtSr0rrUScovjQau7ye4Txb7NwvyVGbtq2XIvFkAKERwGE6NN
MEswDgYD\nVR0PAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwKwYDVR0jBCQwIoAg5XfR/yFYvzNX\nMIJ1pp+uieVXLg5lpKRLCq1R7GFRiwAwCgYIKoZIzj0EAwIDRwAwRAIgSmMPfroG\nNfeh/nSq6hP1ksbG8/47ctew9iZ2
Mef7C4gCIAmtAegh+29gqXD2FskL5o2+jZBt\neOQ8F0PZUzYIiTG8\n-----END -----\n" signature:"0E\002!\000\201v\255\356\306aq\235\364\255\002\262\212"\2408:\326.~\377\005\222\274pi
"\215\177\034s]\002 r`.\376#NG\035\330\r\024\367\345g.*gZA+3>$\254\203\251\373\371:Tv" >
2017-09-07 15:40:08.552 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200 payload:"Draft"
2017-09-07 15:40:08.554 UTC [main] main -> INFO 00b Exiting.....
```

现在的状态是 Draft，然后我们试一试提交报销单 EXP1。

```
    peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile
$ORDERER_CA -C mychannel -n fsmtest -c '{"Args":["Submit","EXP1"]}'
```

结果如下：

```
root@3711df2a20c4:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile $ORDERER_CA -C mychannel -n f
smtest -c '{"Args":["Submit","EXP1"]}'
2017-09-07 15:40:18.659 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-09-07 15:40:18.659 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-09-07 15:40:18.680 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-09-07 15:40:18.680 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-09-07 15:40:18.684 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6A080312A0C08E2D0C5CD0510...1A0E0A065375626D69740A0445585031
2017-09-07 15:40:18.684 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 09E03BAC97A875CEE3B490F9ABB4C40F7007061B8EAEF8F63EE93E506E182E400
2017-09-07 15:40:18.718 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A94070A6A080312A0C08E2D0C5CD0510...1F9493E63A4CFFA3902D2CEEEB3DC10D
2017-09-07 15:40:18.718 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: BC3310DD37589D3A371350A472D20982EDED220807F07C69618B890F74829DA88
2017-09-07 15:40:18.723 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009 ESCC invoke result: version:1 response:<status:200 message:"OK" payload:"Submited" > payload
:"\n \032S\312_Y\256?\034+\003\275\305.Fv\033\353np\271\371\245"\ok\210qS\nYH\022e\nD\022)\n\007fsmtest\022\036\n\n\n\004EXP1\022\002\010\005\032\020\n\004EXP1\032\010Su
bmited\022ZUZ/\n\004Isco\022ZU1/\n\r\n\00/tsmtest\022ZU2\010\004\032\r\010\310\001\032\010Submited\"\016\022ZU0/tsmtest\032\0031.0" endorsement:<endorser:"\n\00/Org1MSP\
022\374\005-----BEGIN -----\nMIICGDCCAb+gAwIBAgIQIf2jXWyVCUdAc7cHqjDxTDAKBggqhkjOPQQDAjBzMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNU2FuIEZy\nYW5jaXNjb
zEZMBoGA1UEChMQb3JnMS5leGFtcGxlLmNvbTEoMBoGA1UEAxMTY2Eu\nb3JnMS5leGFtcGxlLmNvbTAeFw0xNzA5MDcxNTM3NTJaFw0yNzA5MDUxNTM3NTJa\nMFsxCzAJBgNVBAYTAlVTMRMwEQYDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1T\nYW4gRnJhbmNpc2NvMR8wHQYDVQQDExZwZWYyMC5vcmxLmV4YW1wbGUuY29tMFkw\nEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEY+g+r9cO+3m15XpMq7MkfS6GGMunWspB\n9g6f229RFtSr0rrUScov
jQau7ye4Txb7NwvyVGbtq2XIvFkAKERwGE6NNMEswDgYD\nVROPAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwKwYDVR0jBCQwIoAg5XfR/yFYvzNX\nMIJ1pp+uieVXLg5lpKRLCq1R7GFRiwAwCgYIKoZIzj0EAwIDRwAwRAIgS
mMPfroG\nNfeh/nSq6hP1ksbG8/47ctew9iZ2Mef7C4gCIAmtAegh+29gqXD2FskL5o2+jZBt\neOQ8F0PZUzYIiTG8\n-----END -----\n" signature:"0E\002!\000\346z:m\025\307\021\203\300pUY\266\30
69\263@\031\250\317\031\310\235j\332\003\262\314\311\003N\336\002 o\266\007'\362\023\023\354\320\243\267\307\204L\341\272\037\224\223\346:L\377\243\220-.\356\350=\301\r"
>
2017-09-07 15:40:18.723 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200 payload:"Submited"
2017-09-07 15:40:18.724 UTC [main] main -> INFO 00b Exiting.....
```

我们看到状态已经改为 Submited 了。接下来我们进一步执行一级审批，二级审批，都是执行相同的命令。

```
    peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile
$ORDERER_CA -C mychannel -n fsmtest -c '{"Args":["Approve","EXP1"]}'
```

结果如下：

仅供非商业用途或交流学习使用

```
root@3711df2a20c4:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile $ORDERER_CA -C mychannel -n f
smtest -c '{"Args":["Approve","EXP1"]}'
2017-09-07 15:40:48.264 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-09-07 15:40:48.265 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-09-07 15:40:48.277 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-09-07 15:40:48.277 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-09-07 15:40:48.277 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6A08031A0C0880D1C5CD0510...0F0A07417070726F76650A0445585031
2017-09-07 15:40:48.277 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 7614CCF769D31B1E7D2C1349F434B9EA1213967015E548FAB3753C9514B3C70A3
2017-09-07 15:40:48.294 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A94070A6A08031A0C0880D1C5CD0510...AA5FA6A392683EE9327AA3220551DBB6
2017-09-07 15:40:48.294 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: 59DAE1DF9DAFB083A5E2734CFA5D4C40350596D63997D0DB175168D6657B031C
2017-09-07 15:40:48.303 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009 ESCC invoke result: version:1 response:<status:200 message:"OK" payload:"L1Approved" > paylo
ad:"\n \"\223\224\221\317!\321\236Lj*\371\252\323\364\233a\235\273\240\267\206\225*\007q\301.$\343]+\022i\nF\022+\n\007fsmtest\022 \n\n\n\004EXP1\022\002\010\006\032\022\
n\004EXP1\032\nL1Approved\022\027\n\004Isco\022\017\n\r\n\007fsmtest\022\002\010\004\032\017\010\310\001\032\nL1Approved\"\016\022\007fsmtest\032\0031.0" endorsement:<end
orser:"\n\007Org1MSP\022\374\005-----BEGIN -----\nMIICGDCCAb+gAwIBAgIQIf2jXWyVCUdAc7cHqjDxTDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBx
MNU2FuIEZy\nYW5jaXNjbzEZMBcGA1UEChMQb3JnMS5leGFtcGxlLmNvbTEcMBoGA1UEAxMTY2Eu\nb3JnMS5leGFtcGxlLmNvbTAeFw0xNzA5MDoxNTM3NTJaFw0yNzA5MDUxNTM3NTJa\nMFsxCzAJBgNVBAYTAlVTMRMwEQ
YDVQQIEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1TYnYW4gRnJhbmNpc2NvMR8wHQYDVQQDExZwZWVyMC5vcmoxLmV4YW1wbGUuY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEY+g+r9cO+3m15XpMq7MkfS6GGMunWspB
\n9g6f229RFtSr0rrUSovjQau7ye4Txb7NwvyVGbtq2XIvFkAKERwGE6NNMEswDgYD\nVROPAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwKwYDVR0jBCQwIoAg5XfR/yFYvzNX\nMIJ1pp+uieVXLg5lpKRLCq1R7GFRiwAwCgYI
KoZIzjOEAwIDRwAwRAIgSmMPfroG\nNfeh/nSq6hP1ksbG8/47ctew9iZ2Mef7C4gCIAmtAegh+29gqXD2FskL5o2+jZBt\neOQ8FOPZUzYIiTG8\n-----END -----\n" signature:"0D\002 $E\212\336~k~T\027\0
35.Hd\r\376\232\327\245\306P\255\202\261\320\002E\325\326\276MjW\002 \177\000'\202\264\364Ef7[0\316\322\001\246\"\252_\246\243\222h\)\3512r\243\"\0050\333\266" >
2017-09-07 15:40:48.303 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200 payload:"L1Approved"
2017-09-07 15:40:48.303 UTC [main] main -> INFO 00b Exiting.....
root@3711df2a20c4:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile $ORDERER_CA -C mychannel -n f
smtest -c '{"Args":["Approve","EXP1"]}'
2017-09-07 15:40:51.864 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-09-07 15:40:51.865 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-09-07 15:40:51.872 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-09-07 15:40:51.872 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-09-07 15:40:51.873 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A94070A6A08031A0C0883D1C5CD0510...0F0A07417070726F76650A0445585031
2017-09-07 15:40:51.873 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: 15E06F35AE1CE0E643A6F0F914357E4FD12F77BB9FE553881380D4956611652D
2017-09-07 15:40:51.926 UTC [msp/identity] Sign -> DEBU 007 Sign: plaintext: 0A94070A6A08031A0C0883D1C5CD0510...70CBB1D716122AF4FF7F6FDC48068A12
2017-09-07 15:40:51.926 UTC [msp/identity] Sign -> DEBU 008 Sign: digest: FC26417B9A0F54EE1E8F9B548DA9B762C17D38554454 39E38517024BCD32197
2017-09-07 15:40:51.933 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> DEBU 009 ESCC invoke result: version:1 response:<status:200 message:"OK" payload:"Complete" > payload
:"\n %\022\365X\242\242F\351\210\365U^\227\302=:<\307e\246\3050k\216&7\274\344\345\206\363\322\022e\nD\022)\n\007fsmtest\022\036\n\n\n\004EXP1\022\002\010\007\032\020\n\0
04EXP1\032\010Complete\022\027\n\004Isco\022\017\n\r\n\007fsmtest\022\002\010\004\032\r\010\310\001\032\010Complete\"\016\022\007fsmtest\032\0031.0" endorsement:<endorser
:"\n\007Org1MSP\022\374\005-----BEGIN -----\nMIICGDCCAb+gAwIBAgIQIf2jXWyVCUdAc7cHqjDxTDAKBggqhkjOPQQDAjBzMQsw\nCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQGA1UEBxMNU2F
uIEZy\nYW5jaXNjbzEZMBcGA1UEChMQb3JnMS5leGFtcGxlLmNvbTAeFw0xNzA5MDoxNTM3NTJaFw0yNzA5MDUxNTM3NTJa\nMFsxCzAJBgNVBAYTAlVTMRMwEQYDVQQ
IEwpDYWxpZm9ybmlhMRYwFAYDVQQHEw1TYnYW4gRnJhbmNpc2NvMR8wHQYDVQQDExZwZWVyMC5vcmoxLmV4YW1wbGUuY29tMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEY+g+r9cO+3m15XpMq7MkfS6GGMunWspB\n9g6
f229RFtSr0rrUSovjQau7ye4Txb7NwvyVGbtq2XIvFkAKERwGE6NNMEswDgYD\nVROPAQH/BAQDAgeAMAwGA1UdEwEB/wQCMAAwKwYDVR0jBCQwIoAg5XfR/yFYvzNX\nMIJ1pp+uieVXLg5lpKRLCq1R7GFRiwAwCgYIKoZIz
jOEAwIDRwAwRAIgSmMPfroG\nNfeh/nSq6hP1ksbG8/47ctew9iZ2Mef7C4gCIAmtAegh+29gqXD2FskL5o2+jZBt\neOQ8FOPZUzYIiTG8\n-----END -----\n" signature:"0E\002!\000\2452\324\203\344\026
\277\347\255\265\014\235\272\270\303\353$4.\274\244I=8\261\23575\245U\233\312\002 \005 T0\202Y\244\0365\343\3309\n\240\2618p\313\261\327\026\022*\364\377\177o\334H\006\21
2\022" >
2017-09-07 15:40:51.934 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200 payload:"Complete"
2017-09-07 15:40:51.934 UTC [main] main -> INFO 00b Exiting.....
```

可以看到，这个时候，状态已经是 Complete 了。如果我们再次调用 Approve 函数会怎么样？因为我们在状态机中并没有定义这么一个流转事件，所以肯定报错，无法正常执行的。

```
2017-09-07 16:28:25.119 UTC [chaincodeCmd] chaincodeInvokeOrQuery -> INFO 00a Chaincode invoke successful. result: status:200 payload:"Complete"
2017-09-07 16:28:25.119 UTC [main] main -> INFO 00b Exiting.....
root@bfb1e4cf03c4:/opt/gopath/src/github.com/hyperledger/fabric/peer# peer chaincode invoke -o orderer.example.com:7050  --tls true --cafile $ORDERER_CA -C mychannel -n f
smtest -c '{"Args":["Approve","EXP1"]}'
2017-09-07 16:28:35.066 UTC [msp] GetLocalMSP -> DEBU 001 Returning existing local MSP
2017-09-07 16:28:35.066 UTC [msp] GetDefaultSigningIdentity -> DEBU 002 Obtaining default signing identity
2017-09-07 16:28:35.071 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 003 Using default escc
2017-09-07 16:28:35.071 UTC [chaincodeCmd] checkChaincodeCmdParams -> INFO 004 Using default vscc
2017-09-07 16:28:35.072 UTC [msp/identity] Sign -> DEBU 005 Sign: plaintext: 0A97070A6908031A0B08B3E7C5CD0510...0F0A07417070726F76650A0445585031
2017-09-07 16:28:35.072 UTC [msp/identity] Sign -> DEBU 006 Sign: digest: E93F3F229368B1296E90ACD1F22E5B539FF235825060360C6196BFCAD6216D54
Error: Error endorsing invoke: rpc error: code = Unknown desc = chaincode error (status: 500, message: Current status is Complete does not support event:Approve) - <nil>
Usage:
  peer chaincode invoke [flags]

Flags:
  -C, --channelID string   The channel on which this command should be executed (default "testchainid")
  -c, --ctor string        Constructor message for the chaincode in JSON format (default "{}")
  -n, --name string        Name of the chaincode

Global Flags:
      --cafile string              Path to file containing PEM-encoded trusted certificate(s) for the ordering endpoint
      --logging-level string       Default logging level and overrides, see core.yaml for full syntax
  -o, --orderer string             Ordering service endpoint
      --test.coverprofile string   Done (default "coverage.cov")
      --tls                        Use TLS when communicating with the orderer endpoint
  -v, --version                    Display current version of fabric peer server

root@bfb1e4cf03c4:/opt/gopath/src/github.com/hyperledger/fabric/peer#
```

大家如果也在做这个实验，也可以去测试 Reject 函数，会得到想要的结果的。

（5）总结

总的来说，在 Fabric 的 ChainCode 开发中，引入第三方的库可以方便我们编写更强大的链上代码。而这个 FSM 虽然简单，但是也可以很好地将状态流转的逻辑进行集中，避免了在状态流转时编写大量的 Ugly 的代码。让我们在每个函数中更专注于业务逻辑，而不是

仅供非商业用途或交流学习使用

麻烦的状态转移。

11. chaincode-fabcar 分析

chaincode-fabcar 是 Fabric 官方提供的一个典型链上代码示例。下面从几个方面进行分析。

（1）chaincode 主要框架结构

1）引入了 4 个程序库，用于格式化、处理字节、读取和写入 JSON，以及字符串操作。两个 Hyperledger Fabric 特定的智能合同库。shim 包提供了一些 API，以便 chaincode 与底层区块链网络交互来访问状态变量、交易上下文、调用方证书和属性，并调用其他 chaincode 和执行其他操作。

```
import (
    "bytes"
    "encoding/json"
    "fmt"
    "strconv"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    sc "github.com/hyperledger/fabric/protos/peer"
)
```

2）定义 SmartContract 结构体，然后在该 struct 上定义两个函数 Init 和 Invoke。

```
type SmartContract struct {
}
```

3）定义具有 4 个变量的汽车结构体，结构体标签被编码 /Json 库使用。

```
type Car struct {
    Make   string `json:"make"`
    Model  string `json:"model"`
    Colour string `json:"colour"`
    Owner  string `json:"owner"`
}
```

4）当智能合同 fabcar 由区块链网络实例化时，Init 方法被调用。在 Go 中通过给函数标明所属类型，来给该类型定义方法。下面的 s *SmartContract 即表示给 SmartContract 声明了一个方法。在调用 Init 和 Invoke 的时候，都会传入参数 stub shim.ChaincodeStubInterface，这个参数提供的接口为我们编写 ChainCode 的业务逻辑提供了大量的实用方法。假设一切顺利，将返回一个表示初始化已经成功的 sc.Response 对象。

```
func (s *SmartContract) Init(APIstub shim.ChaincodeStubInterface) sc.Response
{
    return shim.Success(nil)
}
```

5）应用程序请求运行智能合约 fabcar 后，Invoke 方法被调用。在调用 Invoke 的时候，由传入的参数来决定我们具体调用了哪个方法，所以需要用 GetFunctionAndParameters 解析调用时传入的参数。

仅供非商业用途或交流学习使用

```
    func (s *SmartContract) Invoke(APIstub shim.ChaincodeStubInterface)
sc.Response {
        //检索请求的智能合约的函数和参数，GetFunctionAndParameters() (string, []string)
        //将字符串数组的参数分为两部分，数组第一个字是 Function，剩下的都是 Parameter
        function, args := APIstub.GetFunctionAndParameters()
        //找到合适的处理函数，以便与账本进行交互
        if function == "queryCar" {
            return s.queryCar(APIstub, args)
        } else if function == "initLedger" {
            return s.initLedger(APIstub)
        } else if function == "createCar" {
            return s.createCar(APIstub, args)
        } else if function == "queryAllCars" {
            return s.queryAllCars(APIstub)
        } else if function == "changeCarOwner" {
            return s.changeCarOwner(APIstub, args)
        }
        return shim.Error("Invalid Smart Contract function name.")
    }
```

6）主函数仅与单元测试模式相关，任何 Go 程序的起点都是 main 函数，在这里只是为了完整性起见。

```
    func main() {
        //创建了新的智能合同，并向对等节点注册它
        err := shim.Start(new(SmartContract))
        if err != nil {
            fmt.Printf("Error creating new Smart Contract: %s", err)
        }
    }
```

（2）对 State DB（状态数据库）的增删改查

1）查询某辆汽车。

通过 GetState(key string) ([]byte,error) 查询数据。因为我们是 Key Value 数据库，所以根据 Key 来对数据库进行查询是一件很常见、很高效的操作，返回的数据是 byte 数组。

```
    func (s *SmartContract) queryCar(APIstub shim.ChaincodeStubInterface, args []
string) sc.Response {

        if len(args) != 1 {
            return shim.Error("Incorrect number of arguments. Expecting 1")
        }
        carAsBytes, _ := APIstub.GetState(args[0])
        return shim.Success(carAsBytes)
    }
```

2）初始化账本。

通过 PutState(key string, value []byte) error 增改数据，对于 State DB 来说，增加和修改数据是统一的操作，因为 State DB 是一个 Key Value 数据库，如果我们指定的 Key 在数

仅供非商业用途或交流学习使用

据库中已经存在，就是修改操作，如果 Key 不存在，就是插入操作。对于实际的系统来说，我们的 Key 可能是单据编号，或者系统分配的自增 ID+ 实体类型作为前缀，而 Value 则是一个对象经过 JSON 序列号后的字符串。

```
    func (s *SmartContract) initLedger(APIstub shim.ChaincodeStubInterface)
sc.Response {
        cars := []Car{
            Car{Make: "Toyota", Model: "Prius", Colour: "blue", Owner: "Tomoko"},
            Car{Make: "Ford", Model: "Mustang", Colour: "red", Owner: "Brad"},
            Car{Make: "Hyundai", Model: "Tucson", Colour: "green", Owner: "Jin Soo"},
            Car{Make: "Volkswagen", Model: "Passat", Colour: "yellow", Owner: "Max"},
            Car{Make: "Tesla", Model: "S", Colour: "black", Owner: "Adriana"},
            Car{Make: "Peugeot", Model: "205", Colour: "purple", Owner: "Michel"},
            Car{Make: "Chery", Model: "S22L", Colour: "white", Owner: "Aarav"},
            Car{Make: "Fiat", Model: "Punto", Colour: "violet", Owner: "Pari"},
            Car{Make: "Tata", Model: "Nano", Colour: "indigo", Owner: "Valeria"},
            Car{Make: "Holden", Model: "Barina", Colour: "brown", Owner: "Shotaro"},
        }
        i := 0
        for i < len(cars) {
            fmt.Println("i is ", i)
            carAsBytes, _ := json.Marshal(cars[i])
            APIstub.PutState("CAR"+strconv.Itoa(i), carAsBytes)
            fmt.Println("Added", cars[i])
            i = i + 1
        }
        return shim.Success(nil)
    }
```

3）创建汽车。

把对象转换为 JSON 的方法（函数）为 json.Marshal()，也就是说，这个函数接收任意类型的数据 v，并转换为字节数组类型，返回值就是我们想要的 JSON 数据和一个错误代码。当转换成功的时候，这个错误代码为 nil。

```
    func (s *SmartContract) createCar(APIstub shim.ChaincodeStubInterface, args []
string) sc.Response {
        if len(args) != 5 {
            return shim.Error("Incorrect number of arguments. Expecting 5")
        }
        var car = Car{Make: args[1], Model: args[2], Colour: args[3], Owner: args[4]}
        carAsBytes, _ := json.Marshal(car)
        APIstub.PutState(args[0], carAsBytes)
        return shim.Success(nil)
    }
```

4）查询所有汽车。

Key 区间查询 GetStateByRange(startKey, endKey string) (StateQueryIteratorInterface, error) 提供了对某个区间的 Key 进行查询的接口，适用于任何 State DB。由于返回的是一个

仅供非商业用途或交流学习使用

StateQueryIteratorInterface（迭代器）接口，我们需要通过这个接口再做一个 for 循环，才能读取返回的信息。所以我们可以独立出一个方法，专门将该接口返回的数据以 string 的 byte 数组形式返回。

```
    func (s *SmartContract) queryAllCars(APIstub shim.ChaincodeStubInterface)
sc.Response {
        startKey := "CAR0"
        endKey := "CAR999"
        resultsIterator, err := APIstub.GetStateByRange(startKey, endKey)
        if err != nil {
            return shim.Error(err.Error())
        }
        //defer 关键字用来标记最后执行的 Go 语句，一般用在资源释放、关闭连接等操作中，
        //会在函数关闭前调用。多个 defer 的定义与执行类似于栈的操作：先进后出，最先定义的最后执行
        defer resultsIterator.Close()
        //buffer 是一个包含查询结果的 JSON 数组，bytes.buffer 是一个缓冲 byte 类型的缓冲器，
        //存放的都是 byte，这样可直接定义一个 Buffer 变量，而不用初始化
        var buffer bytes.Buffer
        buffer.WriteString("[")
        bArrayMemberAlreadyWritten := false
        //迭代器的两个方法，hasNext：没有指针下移操作，只是判断是否存在下一个元素；
        //next：指针下移，返回该指针所指向的元素
        for resultsIterator.HasNext() {
            queryResponse, err := resultsIterator.Next()
            if err != nil {
                return shim.Error(err.Error())
            }
            //在数组成员前加一个逗号
            if bArrayMemberAlreadyWritten == true {
                buffer.WriteString(",")
            }
            buffer.WriteString("{\"Key\":")
            buffer.WriteString("\"")
            buffer.WriteString(queryResponse.Key)
            buffer.WriteString("\"")
            buffer.WriteString(", \"Record\":")
            //Record 是一个 JSON 对象，所以我们按原样写
            buffer.WriteString(string(queryResponse.Value))
            buffer.WriteString("}")
            bArrayMemberAlreadyWritten = true
        }
        buffer.WriteString("]")
        fmt.Printf("- queryAllCars:\n%s\n", buffer.String())
        return shim.Success(buffer.Bytes())
    }
```

5）改变汽车拥有人。

Unmarshal 是用于反序列化 JSON 的函数，根据 data 将数据反序列化到传入的对象中。

```
func (s *SmartContract) changeCarOwner(APIstub shim.ChaincodeStubInterface,
```

仅供非商业用途或交流学习使用

```
args []string) sc.Response {
        if len(args) != 2 {
            return shim.Error("Incorrect number of arguments. Expecting 2")
        }
        carAsBytes, _ := APIstub.GetState(args[0])
        car := Car{}
        json.Unmarshal(carAsBytes, &car)
        car.Owner = args[1]
        carAsBytes, _ = json.Marshal(car)
        APIstub.PutState(args[0], carAsBytes)
        return shim.Success(nil)
    }
```

以上就是对官方链代码 fabcar 的分析。

4.4.2　应用开发示例

前面讲解了 Fabric 网络的搭建和 ChainCode 的开发，那么在 ChainCode 开发完毕后，我们就需要使用 Fabric SDK 来开发应用程序了。官方虽然提供了 Node.JS、Java、Go、Python 等多种语言的 SDK，但是由于整个 Fabric 迭代更新比较快，很多 SDK 还不成熟和完善，所以此处采用 Node JS 的 SDK，毕竟这个的功能比较齐全，而且也是官方示例的时候使用的 SDK。

我们基于 example02 这个 ChainCode 已经安装部署，并且测试通过了，接下来只是换用 Node SDK 的方式进行查询和调用。这个应用涉及查询账户余额和转账的基本功能。

1. 环境准备

Node.js 是一个跨平台的语言，可以在 Linux、Windows 和 Mac 上安装。我们在开发的时候可以在 Windows 下开发，最后生产环境一般都是 Linux，我们这里以 Ubuntu 为例。Fabric Node SDK 支持的 Node 版本是 v6，不支持最新的 v8 版本。NodeJS 官方给我们提供了很方便的安装方法，具体文档在 https://nodejs.org/en/download/package-manager/#debian-and-ubuntu-based-linux-distributions。

我们只需要执行以下命令即可安装 NodeJS 的最新 v6 版本：

```
curl -sL https://deb.nodesource.com/setup_6.x | sudo -E bash -
sudo apt-get install -y nodejs
```

安装完成后我们可以使用以下两个命令来查看安装的 Node 版本和 npm 版本。

```
node -v
npm -v
```

只要安装好 node 和 npm，接下来我们就可以进行 Fabric Node SDK Application 的开发了。

2. 编写 package.json 并下载依赖模块

我们首先在当前用户的根目录下建立一个 nodeTest 的文件夹，用于存放我们关于 node

仅供非商业用途或交流学习使用

的相关项目文件，然后在其中新建一个包配置文件 package.json。

```
mkdir ~/nodeTest
cd ~/nodeTest
vi package.json
```

在这个文件中，在 package.json 中放入了以下内容：

```
{
"name": "nodeTest",
"version": "1.0.0",
"description": "Hyperledger Fabric Node SDK Test Application",
"scripts": {
"test": "echo \"Error: no test specified\" && exit 1"
    },
"dependencies": {
"fabric-ca-client": "^1.0.0",
"fabric-client": "^1.0.0"
    },
"author": "Devin Zeng",
"license": "Apache-2.0",
"keywords": [
"Hyperledger",
"Fabric",
"Test",
"Application"
    ]
}
```

最主要的就是 dependencies，这里我们放了 Fabric CA Client 和 Fabric Node SDK 的 Client。编辑并保存好该文件后，可以运行 npm install 命令来下载所有相关的依赖模块。但是由于 npm 服务器在国外，所以下载速度可能会很慢。淘宝网为我们提供了国内的 npm 镜像，使得安装 npm 模块快很多。运行的命令是：

```
npm install --registry=https://registry.npm.taobao.org
```

运行完毕后我们查看一下 nodeTest 目录，可以看到多了一个 node_modules 文件夹。这里存放了使用刚才的命令下载下来的所有依赖包。

3. 编写对 Fabric 的 Query 方法

下面我们新建一个 query.js 文件，开始我们的 Fabric Node SDK 编码工作。由于代码比较长，所以不分步讲，直接在代码中增加注释。完整代码如下：

```
'use strict';

var hfc = require('fabric-client');
var path = require('path');
var sdkUtils = require('fabric-client/lib/utils')
var fs = require('fs');
```

仅供非商业用途或交流学习使用

```
    var options = {
        user_id: 'Admin@org1.example.com',
        msp_id:'Org1MSP',
        channel_id: 'mychannel',
        chaincode_id: 'mycc',
        network_url: 'grpcs://localhost:7051',      //因为启用了 TLS，所以是 grpcs，
                                                    //如果没有启用 TLS，那么就是 grpc
        privateKeyFolder:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/
e2e_cli/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/
msp/keystore',
        signedCert:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/e2e_
cli/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/
msp/signcerts/Admin@org1.example.com-cert.pem',
        tls_cacerts:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/e2e_
cli/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.com/
tls/ca.crt',
        server_hostname: "peer0.org1.example.com"
    };

    var channel = {};
    var client = null;
    const getKeyFilesInDir = (dir) => {
    //该函数用于找到 keystore 目录下的私钥文件的路径
    var files = fs.readdirSync(dir)
    var keyFiles = []
        files.forEach((file_name) => {
            let filePath = path.join(dir, file_name)
    if (file_name.endsWith('_sk')) {
                keyFiles.push(filePath)
            }
        })
    return keyFiles
    }
    Promise.resolve().then(() => {
        console.log("Load privateKey and signedCert");
        client = new hfc();
    var createUserOpt = {
                    username: options.user_id,
                    mspid: options.msp_id,
                    cryptoContent: { privateKey: getKeyFilesInDir(options.
privateKeyFolder)[0],
        signedCert: options.signedCert }
            }
    //以上代码指定了当前用户的私钥、证书等基本信息
    return sdkUtils.newKeyValueStore({
                            path: "/tmp/fabric-client-stateStore/"
                    }).then((store) => {
                            client.setStateStore(store)
    return client.createUser(createUserOpt)
                    })
```

仅供非商业用途或交流学习使用

```
}).then((user) => {
    channel = client.newChannel(options.channel_id);

    let data = fs.readFileSync(options.tls_cacerts);
    let peer = client.newPeer(options.network_url,
        {
            pem: Buffer.from(data).toString(),
'ssl-target-name-override': options.server_hostname
        }
    );
    peer.setName("peer0");
// 因为启用了 TLS，所以上面的代码就是指定 TLS 的 CA 证书
    channel.addPeer(peer);
return;
}).then(() => {
    console.log("Make query");
var transaction_id = client.newTransactionID();
    console.log("Assigning transaction_id: ", transaction_id._transaction_id);
// 构造查询 request 参数
    const request = {
        chaincodeId: options.chaincode_id,
        txId: transaction_id,
        fcn: 'query',
        args: ['a']
    };
return channel.queryByChaincode(request);
}).then((query_responses) => {
    console.log("returned from query");
if (!query_responses.length) {
        console.log("No payloads were returned from query");
    } else {
        console.log("Query result count = ", query_responses.length)
    }
if (query_responses[0] instanceof Error) {
        console.error("error from query = ", query_responses[0]);
    }
    console.log("Response is ", query_responses[0].toString());// 打印返回的结果
}).catch((err) => {
    console.error("Caught Error", err);
});
```

编写完代码，我们想要测试一下我们的代码是否靠谱，直接运行“node query.js”即可。可以看到，a 账户的余额是 90 元。

```
studyzy@ubuntu1:~/nodeTest$ node query.js
Load privateKey and signedCert
Make query
Assigning transaction_id: ee3ac35d40d8510813546a2216ad9c0d91213b8e1bba9b7fe19cfeff3014e38a
returned from query
Query result count =  1
Response is  90
```

仅供非商业用途或交流学习使用

为什么 a 账户余额是 90？因为我们运行 e2e_cli 的 Fabric 网络时，系统会自动安装 example02 的 ChainCode，然后自动运行查询、转账等操作。

4. 编写对 Fabric 的 Invoke 方法

相比较于 Query 方法，Invoke 方法要复杂得多，主要是因为 Invoke 需要与 Orderer 通信，而且发起了 Transaction 之后，还要设置 EventHub 来接收消息。下面给出 invoke.js 的全部内容，对于比较重要的部分进行了注释。

```
'use strict';

var hfc = require('fabric-client');
var path = require('path');
var util = require('util');
var sdkUtils = require('fabric-client/lib/utils')
const fs = require('fs');
var options = {
    user_id: 'Admin@org1.example.com',
    msp_id:'Org1MSP',
    channel_id: 'mychannel',
    chaincode_id: 'mycc',
    peer_url: 'grpcs://localhost:7051',  // 因为启用了 TLS，所以是 grpcs，
                                         // 如果没有启用 TLS，那么就是 grpc
    event_url: 'grpcs://localhost:7053', // 因为启用了 TLS，所以是 grpcs，
                                         // 如果没有启用 TLS，那么就是 grpc
    orderer_url: 'grpcs://localhost:7050', // 因为启用了 TLS，所以是 grpcs，
                                         // 如果没有启用 TLS，那么就是 grpc
    privateKeyFolder:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/
e2e_cli/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.
com/msp/keystore',
    signedCert:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/e2e_
cli/crypto-config/peerOrganizations/org1.example.com/users/Admin@org1.example.com/
msp/signcerts/Admin@org1.example.com-cert.pem',
    peer_tls_cacerts:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/
e2e_cli/crypto-config/peerOrganizations/org1.example.com/peers/peer0.org1.example.
com/tls/ca.crt',
    orderer_tls_cacerts:'/home/studyzy/go/src/github.com/hyperledger/fabric/examples/
e2e_cli/crypto-config/ordererOrganizations/example.com/orderers/orderer.example.
com/tls/ca.crt',
    server_hostname: "peer0.org1.example.com"
};

var channel = {};
var client = null;
var targets = [];
var tx_id = null;
const getKeyFilesInDir = (dir) => {
//该函数用于找到 keystore 目录下的私钥文件的路径
        const files = fs.readdirSync(dir)
        const keyFiles = []
```

仅供非商业用途或交流学习使用

```
            files.forEach((file_name) => {
                    let filePath = path.join(dir, file_name)
    if (file_name.endsWith('_sk')) {
                            keyFiles.push(filePath)
                    }
            })
    return keyFiles
    }
    Promise.resolve().then(() => {
        console.log("Load privateKey and signedCert");
        client = new hfc();
    var createUserOpt = {
                    username: options.user_id,
                    mspid: options.msp_id,
                    cryptoContent: { privateKey: getKeyFilesInDir(options.
privateKeyFolder)[0],
        signedCert: options.signedCert }
            }
    //以上代码指定了当前用户的私钥、证书等基本信息
    return sdkUtils.newKeyValueStore({
                            path: "/tmp/fabric-client-stateStore/"
                    }).then((store) => {
                            client.setStateStore(store)
    return client.createUser(createUserOpt)
                    })
    }).then((user) => {
        channel = client.newChannel(options.channel_id);
        let data = fs.readFileSync(options.peer_tls_cacerts);
        let peer = client.newPeer(options.peer_url,
            {
                pem: Buffer.from(data).toString(),
    'ssl-target-name-override': options.server_hostname
            }
        );
    //因为启用了 TLS，所以上面的代码就是指定 Peer 的 TLS 的 CA 证书
        channel.addPeer(peer);
    //接下来连接 orderer 的时候也启用了 TLS，也是同样的处理方法
        let odata = fs.readFileSync(options.orderer_tls_cacerts);
        let caroots = Buffer.from(odata).toString();
    var orderer = client.newOrderer(options.orderer_url, {
    'pem': caroots,
    'ssl-target-name-override': "orderer.example.com"
        });

        channel.addOrderer(orderer);
        targets.push(peer);
    return;
    }).then(() => {
        tx_id = client.newTransactionID();
        console.log("Assigning transaction_id: ", tx_id._transaction_id);
    //发起转账行为，从 a 向 b 转账 10 元
```

```
    var request = {
            targets: targets,
            chaincodeId: options.chaincode_id,
            fcn: 'invoke',
            args: ['a', 'b', '10'],
            chainId: options.channel_id,
            txId: tx_id
        };
    return channel.sendTransactionProposal(request);
    }).then((results) => {
    var proposalResponses = results[0];
    var proposal = results[1];
    var header = results[2];
        let isProposalGood = false;
    if (proposalResponses && proposalResponses[0].response &&
            proposalResponses[0].response.status === 200) {
            isProposalGood = true;
            console.log('transaction proposal was good');
        } else {
            console.error('transaction proposal was bad');
        }
    if (isProposalGood) {
            console.log(util.format(
    'Successfully sent Proposal and received ProposalResponse: Status - %s, message
- "%s", metadata - "%s", endorsement signature: %s',
                proposalResponses[0].response.status, proposalResponses[0].response.
message,
                proposalResponses[0].response.payload, proposalResponses[0].endorsement.
signature));
    var request = {
                proposalResponses: proposalResponses,
                proposal: proposal,
                header: header
            };
    var transactionID = tx_id.getTransactionID();
    var eventPromises = [];
            let eh = client.newEventHub();
    //接下来设置EventHub，用于监听Transaction是否成功写入，这里也是启用了TLS
            let data = fs.readFileSync(options.peer_tls_cacerts);
            let grpcOpts = {
                pem: Buffer.from(data).toString(),
    'ssl-target-name-override': options.server_hostname
            }
            eh.setPeerAddr(options.event_url,grpcOpts);
            eh.connect();

            let txPromise = new Promise((resolve, reject) => {
                let handle = setTimeout(() => {
                    eh.disconnect();
                    reject();
                }, 30000);
```

```
    // 向 EventHub 注册事件的处理办法
                eh.registerTxEvent(transactionID, (tx, code) => {
                    clearTimeout(handle);
                    eh.unregisterTxEvent(transactionID);
                    eh.disconnect();

    if (code !== 'VALID') {
                        console.error(
    'The transaction was invalid, code = ' + code);
                        reject();
                    } else {
                        console.log(
    'The transaction has been committed on peer ' +
                            eh._ep._endpoint.addr);
                        resolve();
                    }
                });
            });
            eventPromises.push(txPromise);
    var sendPromise = channel.sendTransaction(request);
    return Promise.all([sendPromise].concat(eventPromises)).then((results) => {
                console.log(' event promise all complete and testing complete');
    return results[0];
            }).catch((err) => {
                console.error(
    'Failed to send transaction and get notifications within the timeout period.'
                );
    return 'Failed to send transaction and get notifications within the timeout period.';
            });
        } else {
            console.error(
    'Failed to send Proposal or receive valid response. Response null or status is
not 200. exiting...'
            );
    return 'Failed to send Proposal or receive valid response. Response null or
status is not 200. exiting...';
        }
    }, (err) => {
        console.error('Failed to send proposal due to error: ' + err.stack ? err.stack :
            err);
    return 'Failed to send proposal due to error: ' + err.stack ? err.stack :
            err;
    }).then((response) => {
    if (response.status === 'SUCCESS') {
            console.log('Successfully sent transaction to the orderer.');
    return tx_id.getTransactionID();
        } else {
            console.error('Failed to order the transaction. Error code: ' + response.status);
    return 'Failed to order the transaction. Error code: ' + response.status;
        }
    }, (err) => {
```

```
        console.error('Failed to send transaction due to error: ' + err.stack ? err
            .stack : err);
    return 'Failed to send transaction due to error: ' + err.stack ? err.stack :
            err;
    });
```

保存文件并退出，接下来测试一下我们的代码。

```
node invoke.js
```

可以看到系统返回如下结果：

```
Load privateKey and signedCert
Assigning transaction_id:  1adbf20ace0d1601b00cc2b9dfdd4a431cfff9a13f6a6f5e5e4
a80c897e0f7a8
transaction proposal was good
Successfully sent Proposal and received ProposalResponse: Status - 200,
message - "OK", metadata - "", endorsement signature: 0D x◆◆N◆◆n◆#◆◆◆/◆G◆◆◆
QD◆w◆◆◆◆◆As◆ \] ◆◆FfW◆◆+◆◆◆◆◆◆=m9I◆◆◆◆ 6◆i
info: [EventHub.js]: _connect - options {"grpc.ssl_target_name_
override":"peer0.org1.example.com","grpc.default_authority":"peer0.org1.example.
com"}
The transaction has been committed on peer localhost:7053
    event promise all complete and testing complete
Successfully sent transaction to the orderer.
```

从打印出的结果看，我们的转账已经成功了。我们可以重新调用之前编写的 query.js 重新查询，可以看到 a 账户的余额已经减少了 10 元。

4.5　Fabric 方案设计

4.5.1　数据库选用方案

Fabric 状态数据库支持两种数据库，分别是 LevelDB 和 CouchDB。其中，LevelDB 是默认的状态数据库。两种数据库均支持对 key 的查询、写入基本操作。对于条件查询，使用 LevelDB 的一般做法是将外键或一些字段直接放在 key 中，可以参考文章“Hyperledger-Fabric”中对链码 API 的介绍。在该文章的示例中，将 sex 和 name 这两个字段放进了 key 中，就可以通过 GetStateByPartialCompositeKey 这个方法，根据局部的复合键查询，得到多个 sex 同为“boy”的结果集。但是，这个方法用于复杂的业务场景中显得有些薄弱：在复杂的业务中需要查询的字段很多，按这个方法只能都放在 key 上；GetStateByPartialCompositeKey 的第 2 个参数 attributes，不能够只提供中间一部分的键值。比如，想要查询 name 为“xiaowang”的用户，即 attributes 只提供“xiaowang”，而不知道第 1 个复合键的值，那么这个方法是不行的。因为 GetStateByPartialCompositeKey 内部使用 range query。但是使用 CouchDB，只需要提供相应查询语句即可，如下：

```
{
selector: {
name: "xiaowang"
}
}
```

通过我们的实践发现，在复杂的业务场景中适合使用 CouchDB。以下旨在快速了解 CouchDB 在 Hyerledger Fabric 中的应用，借助部分实战经验帮助读者了解 CouchDB 的概念及搭建流程。以下所有的结果均在 Mac OS 下演示，Fabric 版本号为 1.1.0，CouchDB 镜像版本号为 x86_64-1.1.0-preview。

CouchDB 是一种面向文档的数据库，提供 REST 风格的插入、更新和检索，数据均以键值对的形式存在，以 JSON 的方式存在文件中。主要特性如下：

- 使用 RESTFul API
- 基于 JSON 存储
- 每个数据库对应单个文件
- 快速检索和索引
- 快速完成备份
- 基于文档存储，不用创建对应的表格，数据之间没有关系范式要求
- 读写均不锁库

（1）Fabric 选择 CouchDB

可以看到，CouchDB 是侧重于 AP（可用性和分区容忍度），面向文档的。我们想象一个场景，在区块链项目中，我们只需要存储表单简单的信息，在各数据中没有关系的需求，仅仅需要存储和查询，而且需要快速检索得到结果，且支持分布式。结合 CouchDB 和以上的表，我们应该可以很快理解 Fabric 开发人员选择 CouchDB 的原因。

在开发的过程中，一开始选择同步区块数据 +graphql 的方式实现复杂的查询，但后来发现，在 query 或 invoke 的过程中读取区块高度时，query 和 invoke 都会卡住，导致 timeout，以致系统无法正常使用。所以最后放弃了同步方案，而采用 CouchDB 原生支持的富查询。

（2）CouchDB 的配置和使用

1）下载 CouchDB 镜像。

```
docker pull hyperledger/fabric-CouchDB:x86_64-1.1.0-preview
```

2）为 Peer 增加 CouchDB 配置。

```
- CORE_LEDGER_STATE_STATEDATABASE=CouchDB
-
CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=CouchDB.peer0.org1.examp
le.com:5984
- CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=YOUR_USERNAME
- CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=YOUR_PASSWORD
```

3）在 docker-compose 文件中加入 CouchDB 配置。

```
CouchDB.peer0.org1.example.com:
container_name: CouchDB.peer0.org1.example.com
image: hyperledger/fabric-CouchDB
# Populate the COUCHDB_USER and COUCHDB_PASSWORD to set an admin
user and password
# for CouchDB. This will prevent CouchDB from operating in an
"Admin Party" mode.
environment:
- COUCHDB_USER= YOUR_USERNAME
- COUCHDB_PASSWORD= YOUR_PASSWORD
ports:
- 5984:5984
network_mode: "service: net_basic"
```

启动以后，我们可以通过 http://localhost:5984/_utils 访问 CouchDB Web 界面。在 Databases 中选择对应的数据库，输入相应查询语句进行查询即可，如图 4-7 所示。

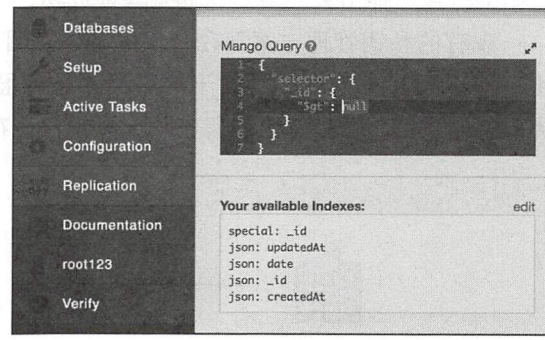


图 4-7　访问 CouchDB Web 界面

在实践中有两个问题：其一是在测试过程中发现用 Fabric 调用 CouchDB 时，使用 limit 和 skip 并不起作用，网上也有开发者反映，网址为：https://stackoverflow.com/questions/49639210/CouchDB-hyperledger-skip-limit-problems。其二是排序问题。对于排序，需要注意的是，不仅要对需要排序的字段创建索引，还必须在 selector 的 key 中包含该字段。

4.5.2　私钥证书管理方案

为了说明 Fabric 的身份认证机制，我们借助一个例子解释几个概念。假设你到超市购买东西，店员让你选择支付方式，他告诉你支持支付宝、微信或者银行卡。你选择了微信支付，微信会验证你的身份，然后完成了这次付款。在这个流程中，店员询问你支付方式就是 PKI（Public Key Infrastructure），它在 Fabric 中有不同类型的验证方法。然后你选择了微信支付，微信确认你是属于微信用户列表内的，这就是 MSP（Membership Service Provider），它验证了你在区块链网络中的身份是否可信。其中 PKI 包含 4 个关键因素：

- 数字证书
- 公钥和私钥
- 证书颁发机构
- 证书吊销列表

MSP 分为两个组织，分别是 ORG1 和 ORG2。Fabric 通过成员来描述主体，其形式为

MSPORG.MEMBER，例如，ORG2-MSP-GOVERNMENT 反映的是 ORG2 组织中 GOVERNMENT 的监管者。有一点需要特别说明的是，在组织中，节点的物理位置在哪并不重要，它可以位于任何一个组织拥有的数据中心或者本地计算机上，但是它相关的身份信息由其组织决定。

以下简要的介绍 Fabric 中关于私钥的流程。首先，通过 Cryptogen 和配置文件 crypto-config.yaml 生成网络实体的加密材料。我们为组织和属于这些组织的组件生成一组证书和密钥。每个组织都配置了唯一的证书，它将特定组件绑定到该组织。通过为每一个组织分配唯一的 CA 证书，网络中的每个成员将使用自己的证书颁发机构。Hyperledger Fabric 中的交易和通信是先通过存储在 keystore 中的实体的私钥签名，再通过公钥进行验证。然后，我们将通过私钥签名的事务请求发送给 peer 节点，peer 节点通过 MSP 和 PKI 验证用户身份，将结果打包广播出去。最后，如果验证成功，背书通过，则此次事务完成。

现有的私钥管理方案存在一定的缺陷。目前 Fabric 官方支持 Go、Nodejs、Java 等几种主流编程语言的 SDK，但对于浏览器并没有很好地支持，所以现有的 Fabric 实现方案中多为服务端代替客户端请求链码，那么私钥只能存储在中心化服务器上，如图 4-8 所示。

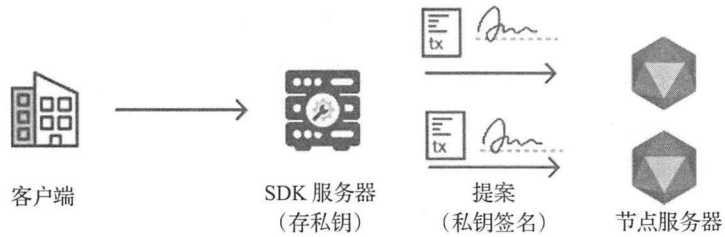


图 4-8 Fabric 私钥管理

显然，在这种模式下，存在着巨大的安全问题，后端开发或运维人员可以轻松地获取用户私钥，伪造用户的身份来完成请求，而客户对此一无所知。我们针对上述方案做了安全方面的改进，将用户的私钥仅保存在客户端，即由用户保存自己的私钥。而链上则存储经用户加密过的私钥，这样即使运维人员也无法获取用户的原始私钥，从而无法伪造用户的身份向 Fabric 发起请求。整个过程大致如下：

1）客户端生成私钥。

2）客户端生成 csr 请求向 Fabric CA 请求证书。

3）客户端收到证书后，将请求数据签名打包后向 Fabric 发起 invoke 或 query 请求。

下面，我们以一个例子（见图 4-9），介绍如何在浏览器端生成签名请求，并直接与 Fabric 进行通信。以下所有的结果均在 Mac OS 下产生，Fabric 版本号为 V1.1.0，相关 docker 镜像版本为 x86_64-1.1.0。演示环境如下：

- 前端框架：React
- 加解密库：jsrsasign
- 服务端语言：Nodejs

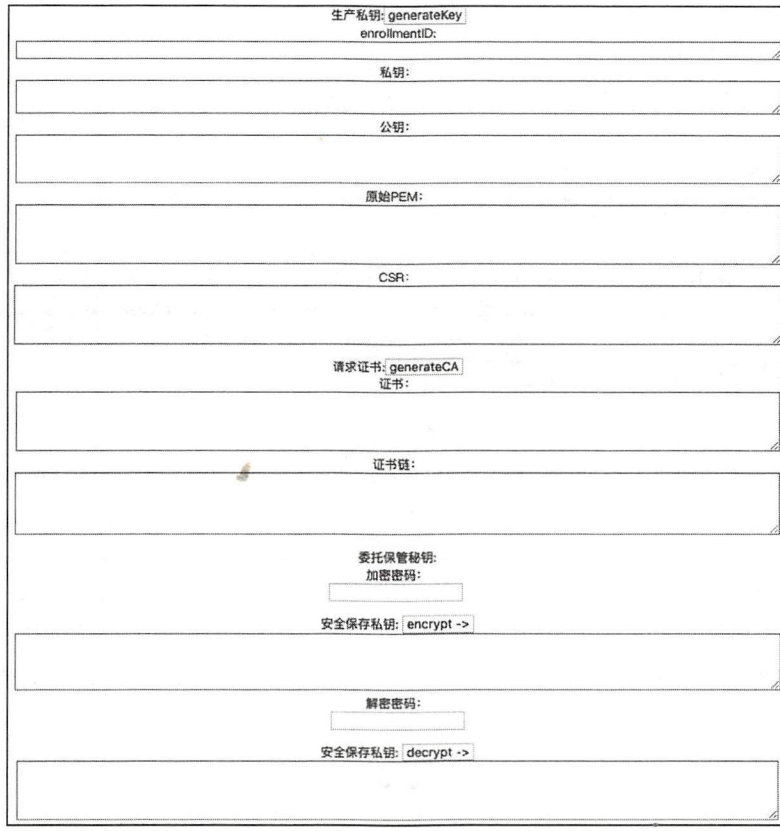
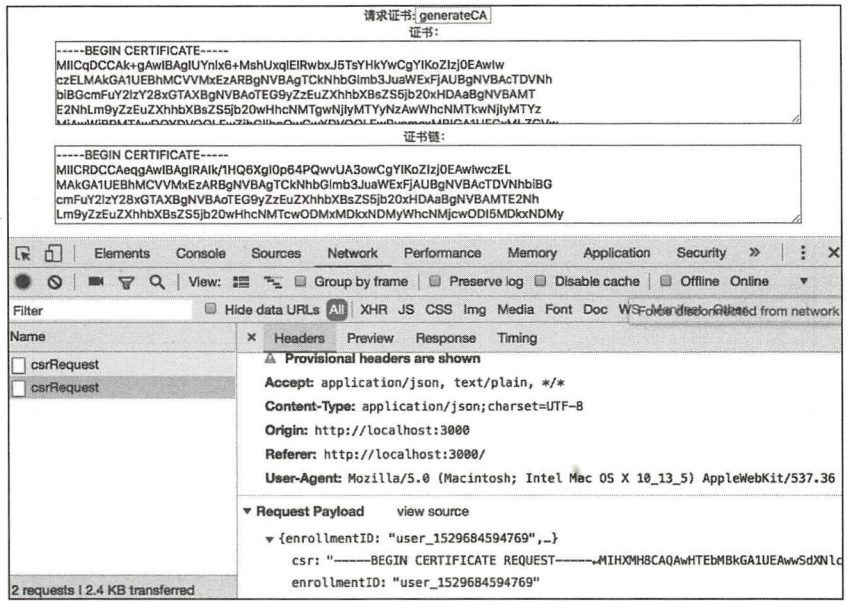


图 4-9　Fabric 签名请求示例

1）通过 jsrsa.KEYUTIL.generateKeypair 生成公私密钥对，再通过 jsrsa.KEYUTIL.getPEM 方法生成原始 PEM，然后通过 jsrsa.asn1.csr.CSRUtil.newCSRPEM 生成 CSR 请求。此时，单击 generateKey 按钮，页面上会显示出我们需要的公私密钥及 CSR 请求。

2）利用 CSR 向 Fabric CA 请求证书。单击按钮，便会显示用户的证书和证书链。

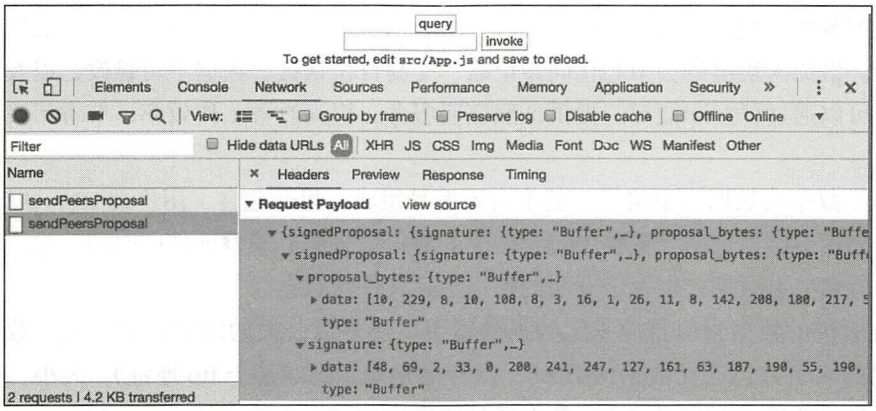


下面，我们通过 jsrsa.KEYUTIL.getPEM 方法利用用户的密码将用户的原始 PEM 进行加密，得到加密后的 PEM（加密后的 PEM 可以安全地存储在区块链中，因为别人不知道用户的密码，无法解开原始 PEM）。

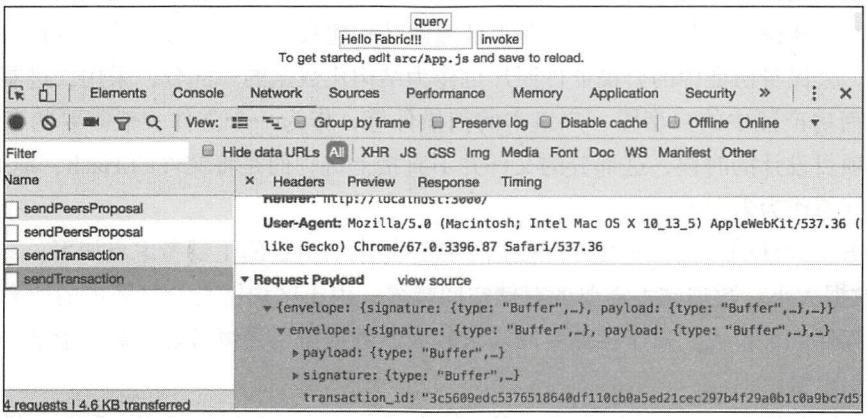
因为用用户的密码进行加密，那么只有通过用户的密码才可以解密。使用上述密码对加密后的 PEM 进行解密。

可以看到，原始 PEM 被成功解密，此时用户可以通过该 PEM 对请求进行签名，就可以正式向 Fabric 进行请求了。

单击 query 按钮，可以看到 network 一栏中已被签名的请求。若切换到控制台，可以看到查询结果为“Hello Fabric!”。



输入“Hello Fabric!!!”，再单击 invoke，在第一个提案请求发出后，得到各节的背书，然后将数据打包发送至 orderer 节点。



至此，我们的 demo 示例便结束了，整个过程大致可以总结为：

1）浏览器生成公私钥及 CSR 请求。

2）浏览器向 Fabric CA 发起证书请求。

3）浏览器用私钥签名请求数据，向 Fabric Peer 进行链码调用。

4）各背书节点返回提案响应，服务器将这些背书打包，发送给 orderer 节点，orderer 对数据进行验证，无问题后写区块数据并广播给各节点，最后返回浏览器响应。

5）浏览器进行查询时，获得提案响应后便展示出相关信息，而不再将请求发送至 orderer 节点，这样，数据便不会写入区块。是否将请求发送给 orderer 节点，也是 query 与 invoke 的唯一区别。

仅供非商业用途或交流学习使用

4.5.3 数据上链方案

在项目实践过程中，数据上链的方案主要有元数据上链和数据索引上链两种方式。为了便于介绍二者的区别，我们通过区块链电子证照共享的案例进行简要说明。该案例主要是以区块链技术实现个人、法人的电子证照数据共享，由城市内政府服务职能部门共同组成城市电子证照数据区块链网络，在原有人口、法人信息共享基础上提供电子证照数据发布、检索等服务。借助区块链的去中心化同步记账、交易身份认证、数据不可篡改，以及数据加密等手段，可以通过任意职能部门提供证照证明服务，提高政务工作效率，节省市民、企业的办事时间。

此处元数据由数据类型编号 + 基本内容编号组成，里面包含了用户的 ID、数据内容等。比如一个公民的结婚证明元数据中包含公民的身份证号码、颁发证件的时间、地点、配偶身份号码、姓名、有效期，以及该证明的扫描图片文件等数据。

此处所讲的数据索引指的是元数据的索引。还是以公民的结婚证明为例，数据索引指该元数据的索引，包含元数据的 Hash、保存的地址（比如某个 ftp 地址）、大小、类型这几个基础信息。数据索引中不包含元数据的具体内容。

在项目实践上，对于轻量级的结构化数据采用元数据上链方案；对于使用频度较高、轻量级的非结构化数据，也采用元数据上链方案；而对于重量级的非结构化数据则采用索引上链的方案。具体说来，公民和法人的各类证照的结构化数据、审批过程的结构化数据直接元数据上链，公民身份证中的公民证件照片也作为结构化数据的一部分，采用二进制码流的形式上链。而房产证中的户型图、户口本的扫描件等，作为一个文件一般需要的存储空间比较大，在项目设计的时候，这部分的文件并不直接上链，而是将文件的 Hash、地址、类型、大小等特征值作为索引上链。

在做方案设计的时候，选择是元数据上链还是索引上链的主要考虑点如下：

1）数据大小。数据越大需要的存储空间越多，由于区块链是全量账本的所有节点全复制，所以存储的代价大。对于存储空间需求较大的数据需要谨慎考虑元数据上链，而优先考虑数据索引上链。

2）数据使用效率。对数据使用效率有较高要求的，需要优先考虑元数据上链。数据索引上链可以节约整个项目的成本，但是在数据的使用过程中，涉及生成索引、通过索引恢复元数据、通过索引校验元数据的过程，无疑损耗了数据的使用效率。我们知道，数据索引包含了元数据的 Hash，我们可以在使用数据的时候验证 Hash，来检验存放在区块链账本上的元数据是否被篡改，校验没有问题的数据可以认为是可信的，如果校验不一致则提示应用异常处理。所以说索引上链也可以保障数据可信性，即使元数据并没有存储在区块链上也可以。

3）数据安全。对数据的安全有较高要求的，需要考虑优先考虑元数据上链。区块链账本是全节点复制的，某一个或者多个节点的异常不会导致数据的丢失，它天然地对数据安全

仅供非商业用途或交流学习使用

有保护作用。当然，不保存在区块链上的数据，如果有自己的一套数据安全保障方案，比如异地的数据容灾方案、主备机的安全方案等，就不一定需要通过全区块链节点的多副本来存储，毕竟存储成本太大。

从以上几个方面看，元数据上链和索引上链这两种方案并没有绝对的优劣，需要根据项目的具体情况做分析，从而设计出适合项目需要的方案。在本项目实践中，实际上是综合了元数据上链和索引上链的两种方式，对不同的数据采用了不同的方案。

4.5.4　背书验证方案

如何解决信任问题是任何区块链实践都面临的关键问题。基于 Fabric 的原理，除了有一般区块链的分布式、链上账本等基本特性外，还有其独特的机制来保障信任。Fabric 共识被定义为一个完整的循环，从提交请求、背书验证，到事务排序、确认和广播。它是由一个经过验证核实的区块所包含的一组事务。简单地说，当一个区块中的事务集合的顺序和结果经过所有检查而符合策略标准时，将最终达成一致。这些检查和平衡发生在一个个请求事务的生命周期中，包括使用背书策略来规定哪些特定的成员必须支持某个事务类，以确保这些策略得到执行和维护。Fabric 的背书策略以及背书验证过程如下。

1. 背书策略

节点通过背书策略来确定一个交易是否被正确背书。当一个 peer 接收一个交易后，就会调用与该交易 chaincode 相关的 VSCC（实例化时指定的）作为交易验证流程的一部分，来确定交易的有效性。为此，一个交易包含一个或多个来自背书节点的背书。VSCC 的任务是验证：

1）所有的背书是有效的（即有效证书进行的有效签名）

2）恰当的（满足要求的）背书数量

3）背书来自预期的背书节点

背书策略即是用于定义 2 和定义 3 的验证规则。

2. CLI 中背书策略语法

在 CLI 中，用一种简单的布尔表达式来表示背书策略。Fabric 使用 MSP 来描述主体，MSP 用于验证签名者的身份和签名者在 MSP 中的角色和权限。目前支持 member、admin、client 和 peer。主体的描述形式是 MSP.ROLE，其中 MSP 是 MSP ID，ROLE 是 member、admin、client 或 peer。比如，一个有效的主体可表示为 Org0.admin（MSP Org0 的管理员）或 Org1.member（MSP Org1 的成员）或 Org0.client（MSP Org0 的客户端）或 Org1.peer（MSP Org1 的节点）。

CLI 语法是：EXPR(E[, E...])。其中 EXPR 是 AND 或 OR，表示布尔表达式；E 是上面语法所描述的主体或者是另一个嵌套进去的 EXPR。例如：

AND('Org1.member', 'Org2.member', 'Org3.member') 表示需要 3 个主体共同签名背书。

仅供非商业用途或交流学习使用

OR('Org1.member', 'Org2.member') 表示需要两个主体之一的签名背书。

OR('Org1.member',AND('Org2.member', 'Org3.member')) 表示需要 Org1 的签名背书或者 Org2 和 Org3 共同的签名背书。

3. 为 chaincode 指定背书策略

如果在实例化 chaincode 时未指定背书策略，则背书策略默认为“由通道中的组织的任何一个成员进行背书”。例如，一个带有“Org1”和“Org2”的通道将有一个默认的背书策略：OR（‘Org1. 成员’，‘Org. 成员’）。

命令 peerchaincode instantiate -C <channelid> -n mycc -P "AND('Org1.peer','Org2.peer')" 实例化 chaincode 后添加到通道中的新组织，可以查询 chaincode，但将无法提交由其背书的事务。必须修改背书策略，来让由新加入的组织背书的交易得到认可。

4. 背书策略的验证过程

1）客户端（SDK）把交易提议（TX Proposal）发给指定的一个或多个背书节点 (endorsing peer)。接收提议的背书节点在 SDK 的交易提议请求中指定，最终进行背书的节点由交易所属的 chaincode 和该 chaincode 所定义的背书策略（Endorsement Policy）共同决定。例如 node.js sdk 的交易提议请求接口如下：

```
Channel. sendTransactionProposal(request, timeout)
```

request 是一个 ChaincodeInvokeRequest 对象，可以在该对象中指定目标节点，如果未指定，则会将交易提议请求发送给加入该通道的所有节点。

2）背书节点收到交易提议后，首先用客户端（SDK）的公钥验证它的签名是否有效、客户端是否可以在该通道进行操作、交易是否已被提交、交易提议组织是否正确。验证通过后模拟执行 chaincode（不会将结果写入账本里），将执行的结果反馈给客户端。

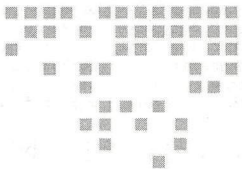
3）客户端（SDK）收到足够多的背书节点的信息后（背书策略），表示这个交易已经正确背书，然后将交易提议、模拟结果和背书信息打包发给 orderer 节点；如果客户端没有搜集到足够多的背书节点反馈的背书信息，这个交易就会被舍弃。

4）orderer 节点对来自客户端（SDK）的信息进行排序，并创建区块，然后在通道上进行广播。通道上的 peer 节点接收到交易区块后，验证背书策略是否满足，然后更新账本。

至此，背书策略的验证过程完成。

Fabric 的背书策略配置是非常灵活的，支持 AND、OR 以及组合的逻辑控制，比如，需要 3 个节点同时背书 AND(1,2,3)，需要节点 1 或者节点 2 之一背书 OR(1，2)，需要节点 1 或者（节点 2 和节点 3 同时）背书 OR(1,AND（2,3))，都可以通过配置背书策略来实现业务需求。安全的做法是提供至少两个节点的背书。

仅供非商业用途或交流学习使用



第 5 章 Chapter 5

Fabric 源代码解析

5.1 概述

在 Fabric 根目录执行如下命令，递归搜索所有 Go 文件中的 main 函数，并将结果导入 func_main.txt 文件。

```
grep "func main" * -r -n --include=*.go > func_main.txt
```

打开 func_main.txt 文件，会发现大约有七八十条记录，且很有规律，如下：

```
1 bddtests/regression/go/ote/ote.go:1052:func main() {
2 common/configtx/tool/configtxgen/main.go:310:func main() {
3 common/tools/cryptogen/main.go:203:func main() {
4 core/comm/testdata/certs/generate.go:243:func main() {
5 core/ledger/kvledger/example/main/example.go:69:func main() {
6 core/ledger/kvledger/marble_example/main/marble_example.go:70:func main() {
7 examples/ccchecker/chaincodes/newkeyperinvoke/newkeyperinvoke.go:64:func main() {
8 examples/ccchecker/main.go:45:func main() {
9 examples/chaincode/chaintool/example02/src/chaincode/chaincode_example02.go:118:func main() {
10 examples/chaincode/go/chaincode_example01/chaincode_example01.go:99:func main() {
11 examples/chaincode/go/chaincode_example02/chaincode_example02.go:194:func main() {
12 examples/chaincode/go/chaincode_example03/chaincode_example03.go:95:func main() {
13 examples/chaincode/go/chaincode_example04/chaincode_example04.go:162:func main() {
14 examples/chaincode/go/chaincode_example05/chaincode_example05.go:203:func main() {
15 examples/chaincode/go/eventsender/eventsender.go:93:func main() {
16 examples/chaincode/go/invokereturnsvalue/invokereturnsvalue.go:134:func main() {
17 examples/chaincode/go/map/map.go:188:func main() {
18 examples/chaincode/go/marbles02/marbles_chaincode.go:104:func main() {
19 examples/chaincode/go/passthru/passthru.go:60:func main() {
20 examples/chaincode/go/sleeper/sleeper.go:119:func main() {
21 examples/chaincode/go/utxo/chaincode.go:99:func main() {
22 examples/dchackfest/samples/e2e/chaincodes/go/chaincode_example02/chaincode_example02.go:194:func main() {
23 examples/dchackfest/samples/e2e/chaincodes/go/marbles02/marbles_chaincode.go:104:func main() {
24 examples/e2e_cli/examples/chaincode/go/chaincode_example02/chaincode_example02.go:194:func main() {
25 examples/events/block-listener/block-listener.go:135:func main() {
26 orderer/main.go:50:func main() {
27 orderer/sample_clients/broadcast_config/client.go:90:func main() {
28 orderer/sample_clients/broadcast_timestamp/client.go:71:func main() {
29 orderer/sample_clients/deliver_stdout/client.go:98:func main() {
30 orderer/sample_clients/single_tx_client/single_tx_client.go:42:func main() {
31 peer/main.go:77:func main() {
```

仅供非商业用途或交流学习使用


```
32 vendor/github.com/DATA-DOG/godog/builder.go:35:func main() {
33 vendor/github.com/DATA-DOG/godog/cmd/godog/main.go:58:func main() {
34 vendor/github.com/docker/docker/pkg/archive/example_changes.go:27:func main() {
35 vendor/github.com/docker/docker/pkg/mflag/example/example.go:26:func main() {
36 vendor/github.com/docker/docker/pkg/namesgenerator/cmd/names-generator/main.go:9:func main() {
37 vendor/github.com/docker/docker/pkg/plugins/pluginrpc-gen/main.go:60:func main() {
38 vendor/github.com/golang/protobuf/proto/lib.go:234:      func main() {
```

可以看到，结果只集中在 bddtests、common、core、examples、orderer、peer、vendor 这几个子目录下。结果可分为三类：可以直接排除掉的、可辅助研究源码的和源码本身。首先，可以直接排除掉 vendor 子目录中的所有结果，因为 vendor 目录下都是 Go 语言使用的第三方库的代码。其次，目录中有 test、example、sample 字眼的，都属于测试和示例范畴的代码，可以用于辅助研究源码，因为有时候看不懂源码，看看例子是怎么用的，可能就知道这个函数是干什么的或者要怎么使用了。最后是源码本身，也是我们要研究的“线头”。经过以上两步的排除，剩下的就是源码。

- `2 common/configtx/tool/configtxgen/main.go:310:func mainc(){`
- `3 common/tools/cryptogen/main.go:203:func main() {`
- `26 orderer/main.go:50:func main() {`
- `31 peer/main.go:77:func main() {`

若操作过 Fabric 文档中 Getting Started 的话，很容易就能推测出 2、3 行这两个 main 函数用于生成 configtxgen 和 cryptogen 这两个工具。31 行的 main 函数用于生成 peer 程序，因为操作交易时在命令行使用到了 peer…。Getting Started 中虽然没有明显使用到关于 orderer 的操作，但既然被单独赋予了 main 函数，就说明其是自成一体的，自成一个服务，最终生成会一个 orderer 程序。最终，以上的这些指向将我们引向了 orderer 和 peer 的 main 函数，也就是研究整个项目源码的“线头”。接下来的源码分析也是先从 peer 的 main 函数这个“线头”开始。

5.1.1 源码中的简拼

能被 Linux Foundation 支持的项目自然不会差。代码只要看几行就知道是很优秀的代码。代码中很多地方都是英语的全拼，有时候不需要看 ReadMe.md 或注释都能知道这个目录或函数的用途。但有些目录中文件名也有用首字母的，知道这些简拼对于顺利阅读代码也是很有帮助的。罗列如下：

```
MSP: Membership service provider 会员服务提供者
BCCSP: blockchain (前两个字母 BC) cryptographic service provider 区块链加密服务提供者
ab: atomic broadcast 原子 (操作) 广播
lscc: lifecycle system chaincode (CC) 生命周期系统链码
Spec: Specification 规格标准，详细说明
KV: key-value 键 - 值对
CDS: ChaincodeDeploymentSpec
CIS: ChaincodeInvocationSpec
mgmt: management
SW: software-based
```

仅供非商业用途或交流学习使用

AB: AtomicBroadcast
GB: genesis block，创世纪块，也就是区块链中的第一个块
CC 或 cc: chaincode
SCC 或 scc: system chaincode
cscc: configer system chaincode
lscc: lifecycle system chaincode
escc: endorser system chaincode
vscc: validator system chaincode
qscc: querier system chaincode
alg: algorithm 算法
mcs: mspMessageCryptoService
mock: 假装，学样子，模仿的意思，基本上是服务于 xxx_test.go 的，即用于测试的
Gossip: 一种使分布节点达到状态最终一致的算法
attr: attribute
FsBlockStore: file system block store
vdb: versioned database 状态数据库
RTEnv: runtime environment 运行环境
pkcs11: pcks#11 一种公匙加密标准，有一套叫作 Cryptoki 的接口，是一组平台设备无关的 API
MCS: mspMessageCryptoService 消息加密服务
sa: SecurityAdvisor
impl: implement 好多处 XXX.go 和 XXXimpl.go 是对应的，前者用于接口或者定义，后者实现该接口或定义
FSM: finite state machine 有限状态机
FS: file system 文件系统
blk: block
cli: command line interface 命令行界面
CFG: FABRIC_CFG_PATH 中的，是 config 的意思
mgr: manager
cpinfo: checkpoint information 检查点信息
DevMode: development mode 开发模式
Reg: register 注册，登记
hdr: header
impl: implement
oid: ObjectIdentifier 对象标识符
ou 或 OU: organizational unit
CRL: certificate revocation list 废除证书列表
prop: proposal 申请，交易所发送的申请
ACL: Access Control List 访问控制列表
rwset: read/write set 读写集
tx，Tx: transaction 交易
CSP: cryptographic service provider 是 BCCSP 的后 3 个字母，加密服务提供者
opt: option 选项
opts: options 多个选项
SKI: 当前证书标识，所谓标识，一般是对公匙进行一下 Hash
AKI: 签署方的 SKI，也就是签署方的公匙标识
HSM: Hardware Security Modules
ks: KeyStore Key 存储，这个 Key 指的是用于签名的公匙和私匙
oid: OBJECT IDENTIFIER 对象身份标识

5.1.2 源码中的惯例

源码中有很多惯例，也是开发大型项目（与语言无关）需要有的优秀习惯。这一点也是

仅供非商业用途或交流学习使用

值得学习的。目前个人发现的惯例有以下几点，对于阅读源码也是很有帮助的。

1）common 目录下是其所在的层级中的公用的代码。A/common，则说明该 common 中的代码在 A 范围中公用；A/B/C/common，则说明该 common 中的代码在 C 目录中公用。

2）mock 目录是用于方便 Go 测试文件（即众多的 XXX_test.go）进行测试所需要的模拟数据、环境等。研究源码的初始阶段可忽略该类目录。

3）XXX.go 与 XXXimpl.go 是定义与实现的配套代码。

4）同一事务分别存在于不同主题下。如 protos 目录下的 peer 与 core 目录下的 peer 都是 peer 相关的代码，但是相关主题的代码却分开放置。

5）标有 no-tls 的说明相关代码未使用安全传输协议（TLS）。

6）util 文件夹中一般都是该层级或该主题源码中具有辅助性、工具性的代码。

5.1.3 源码目录的基本结构

其实关于源码目录的基本结构应该是在探索源码的过程中自然而然逐渐搞明白的。在此只做记录使用，不建议各位看这部分，而是建议各位在阅读源码过程中，不断地翻目录，慢慢弄懂。最开始研究源码，需要关注的也就是如下这些目录了。

- bcssp 加密服务代码目录。
- common 全局公用代码目录。
- core 核心功能代码目录。
- docs 以 .rst 文件为核心，可编译生成文档，说明文档的目录。
- events 事件代码目录，用于生产和消费信息。
- examples 示例目录。
- gossip 本意是绯闻的意思，是一种可达到去中心化，有一定容错能力且可达到最终一致的传播算法。
- msp 会员服务代码目录。
- orderer 可理解为 orderer 目录，orderer 也算是区块链中的专用名词，用于消息的订阅与分发处理。
- protos 原型目录，定义各种原型和生成的对应的 XXX.pb.go 源码。
- vendor 原意是商贩，在此就是存放 Go 中使用的全部的各种第三方包。

5.2 peer 命令结构

5.2.1 peer 目录结构

peer 目录结构十分清晰，一个 main.go 文件，其余文件夹除 common,gossip 外均为子命令集合，有 chaincode、channel、clilogging、node、version 五个，各司其职，供 main.go 整

仅供非商业用途或交流学习使用

合使用。子命令文件夹中，与文件夹名称相同的 .go 文件为主要源码文件，其余的均为按功能划分的动作命令源码。以 node 目录为例，node 作为根命令的一个子命令，在 node.go 中实现，而 node 这个命令又有 start、status、stop 这 3 个动作，去执行不同的任务，分别在对应的 start.go、status.go、stop.go 中实现。注意，start、status、stop 其实也是子命令，是 node 这个子命令的子命令，因为它们是在命令层级中最终去干活的底层的代码，用动作去形容它们更贴切一些。目录结构如下：

- chaincode
- channel
- clilogging
- common
- gossip
- node
  - node.go
  - start.go
  - status.go
  - stop.go
- version
- main.go

5.2.2　第三方包

在 Getting Started 中，无论是在启动 peer 容器时默认执行的命令，还是手工执行交易时在终端窗口所输入的命令，都类有类似的格式，如 peer channel…，peer node…，peer chaincode…。这种“命令 + 子命令 + 选项”的风格，在感觉上毫无违和感。peer 命令主要依赖第三方包 github.com/spf13/cobra，由其组成基本的 peer 命令架构。所以在此简单介绍一下 cobra。

cobra 既是一个用于生成命令行程序的库，也是用来生成程序和命令文件的程序（即在命令行用 cobra 命令进行一系列操作，格式化生成一些使用 cobra 框架的源代码文件，用户可在此基础上进行编程）。目前，peer 源码只将 cobra 当作一个库进行使用。cobra 基本用法如下。

1）创建一个（根）命令对象，其原型为 Command，每个命令都是一个 Command 对象实例。创建命令对象其实就是填充 Command 中的成员的过程。需要注意的是，Command 中的成员还有很多，其中有一批字段名为 *Run、*RunE 形式的成员，其作用与 Run 类似，区别是：运行的时间有先有后，是否被子命令继承，是否返回错误。

```
type Command struct {
    Use string   //命令名称字段，如你在命令行敲的是peer ...，则该字段值就是"peer"
    Short string //短说明字段
    Long string  //详细说明字段
    Run func(cmd *Command, args []string) //该命令所执行的函数
```

仅供非商业用途或交流学习使用


```
    ...
}
RootCmd := &cobra.Command{...}
```

2）如果有需要，对命令添加 Flags，这一点可以简单地理解为命令选项。

```
RootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose output")
RootCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to read from")
```

3）如果有需要，对根命令添加子命令。子命令与根命令在本质是一样的，只是人为地进行级别上的区分。

```
RootCmd.AddCommand(versionCmd)
```

4）运行命令。

```
RootCmd.Execute()
```

由于本章重点在 peer，所以在此只做简单介绍。更详细的使用方法，可在 go doc 或 github.com/spf13/cobra 上学习。其实阅读 Fabric 源码过程中有一个感觉，就是项目的“大神们”选用的第三方库，一般都是既能满足需求，又比较容易学习和上手。

5.2.3 peer 命令结构解析

下面从 peer/main.go 文件开始解析源码。本节旨在解析 peer 的命令结构，因此只会涉及相关的源码，其他部分将会在其他章节中对应分析。对 cobra 的用法稍微熟悉后，很容易就可以看懂 main 函数的构建。peer 目录下的子命令的源码结构非常类似，也基本逃不出上面介绍的关于 cobra 的基本操作。

1）定义一个 mainCmd 命令 var mainCmd = &cobra.Command{…}，该命令填充了 Use、PersistentPreRunE 和 Run 成员。Use 如我们预见的那样被赋值为 peer，PersistentPreRunE 先于 Run 执行，都被赋值了一个匿名函数。因为 mainCmd 只单纯作为根命令，不实现由子命令实现的具体的交易事务，因此实现的只是 PersistentPreRunE 指定的检查、初始化日志系统并缓存配置的功能，和 Run 指定的版本打印、命令帮助功能。

2）生成 mainCmd 对象的命令行标识对象 mainFlags，mainFlags := mainCmd.PersistentFlags()，也就是 peer 命令的选项，并对该标识对象进行维护，增加了 version、logging_level 两个选项。这也对应了其在自身对象中设置 PersistentPreRunE 和 Run 的功能。

3）添加子命令 mainCmd.AddCommand(…)。添加的命令有 version.Cmd()、node.Cmd()、chaincode.Cmd(nil)、clilogging.Cmd(nil)、channel.Cmd(nil) 五个。Cmd() 是每个子命令文件中暴露出的函数，整合了各自的动作命令。

4）启动根命令 mainCmd.Execute()。启动了根命令，也就启动了其下的所有命令。

5.2.4 子命令结构解析

其实读懂了 peer 命令，其余的子命令类推即可。子命令的源码结构是极其相似的，这

仅供非商业用途或交流学习使用

里只以 node 为例。

1）在 node.go 中，定义了一个 node 命令对象，var nodeCmd = &cobra.Command{…}。

2）在 Cmd 函数中，添加了 startCmd()、statusCmd()、stopCmd() 三个函数返回的 start、status、stop 子命令（动作命令），分别实现在 start.go、status.go、stop.go 中。这 3 个命令的源码结构也是基本一致，在此仅以 start 和 start.go 为例。

3）在 start.go 中，首先定义了一个 start 命令对象，var nodeStartCmd = &cobra.Command{…}，其中对 RunE 成员赋值了一个匿名函数，函数体中执行了 serve 函数，这也是该命令最终会调用的函数。serve 函数是一个非常重要、非常复杂的函数。这在前面介绍 Fabric 项目“线头”时提到过，在每个 peer 容器启动后默认执行的就是 peer node start -peer-defaultchain = false 命令，在此处就对接上了。该命令最终调用执行的就是 serve 函数，同时也就是说，serve 函数会做很多准备工作。

peer 命令结构如图 5-1 所示。

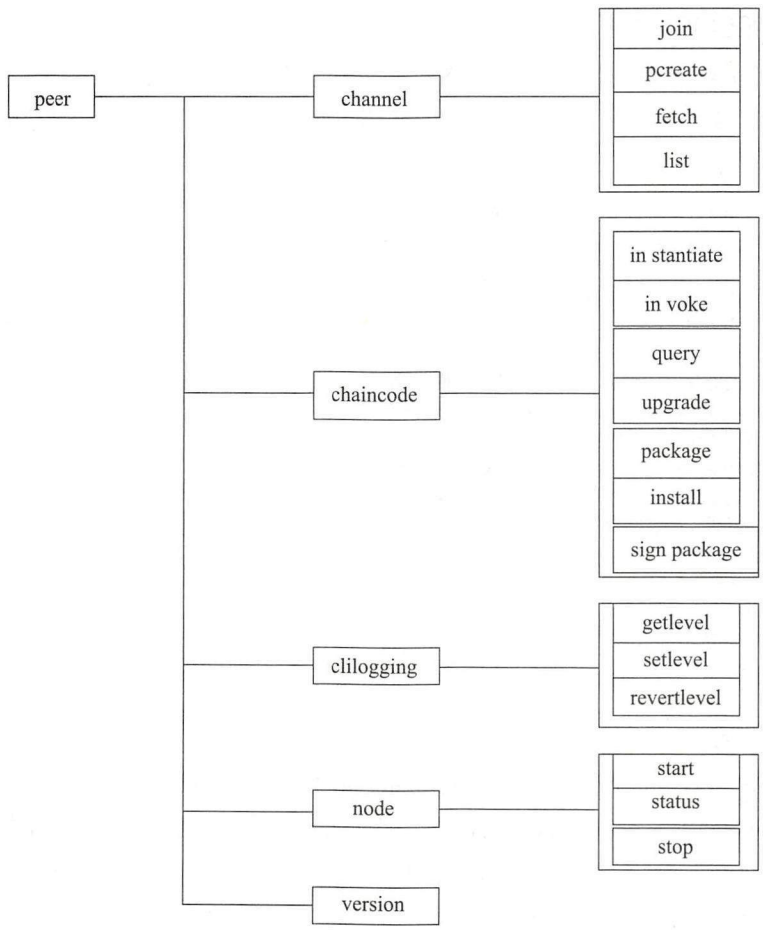


图 5-1　peer 命令结构

仅供非商业用途或交流学习使用

5.3 日志系统

这里所说的日志，是指程序运行过程中打印到终端或日志文件、记录程序运行过程的日志，而非涉及 Fabric 记录交易数据、账本数据一类的交易日志。虽然日志系统机制在进行源码研究时可以忽略且相对简单，但是一方面日志系统对于程序运行和调试必不可少，Fabric 如何形成这个日志系统可以学习学习，日志所打印的内容对阅读源码也是一种提示；另一方面不弄清楚这个日志，在阅读源码的过程中这些到处出现的打印日志总像围绕在源码上面的“蚊子”一样，造成干扰。

Fabric 的日志系统主要使用了第三方包 go-logging，可在 github.com/op/go-logging 下载。很少一部分使用了 Go 语言标准库中的 log。在此基础上 Fabric 自己封装出来了 flogging，这个 f，应该代表 Fabric，意思是说这是 Fabric 的 logging。代码集中在 fabric/common/flogging 目录下，供项目全局使用。

5.3.1 go-logging 简介

logging 其实就是封装的各种打印格式，包括消息层级上的，如 DEBU、消息、注意、警告、错误，以及消息颜色上的，如消息是正常的绿色、错误则是醒目的红色。

基本用法如下：

```
//创建一个名字为examplename的日志对象log
var log = logging.MustGetLogger("examplename")
//创建一个日志输出格式对象format，也就是用什么格式输出
var format = logging.MustStringFormatter(
    `%{color}%{time:15:04:05.000} %{shortfunc} ▶ %{level:.4s} %{id:03x}%{color:reset} %{message}`,
)
//创建一个日志输出对象backend，也就是日志要打印到哪儿，在此是标准错误输出
backend := logging.NewLogBackend(os.Stderr, "", 0)
//将输出格式与输出对象绑定
backendFormatter := logging.NewBackendFormatter(backend, format)
//将绑定了格式的输出对象设置为日志的输出对象
//这样log打印每一句话都会按格式输出到backendFormatter所代表的对象里，在此即是标准错误输出
logging.SetBackend(backendFormatter)
//log打印依据Info信息
log.Info("info")
//log打印一句Error信息
log.Error("err")
```

5.3.2 flogging

在 flogging 目录下有两个文件，grpclogger.go 和 logging.go。

grpclogger.go 用于设置 grpc 的日志，因为 grpc 默认只使用 Go 语言的标准日志接口，因此将 logging 封装成 Go 语言的标准日志形式的结构 type grpclogger struct {logger *logging.Logger}，然后通过 initgrpclogger() 生成对象，供 grpc 使用，从而实现让 grpc 也使用 flogging 的效果。

仅供非商业用途或交流学习使用

logging.go文件中，自带一个名为flogging的日志记录者logger，同时规定了默认的日志格式、日志等级，用defaultFormat、defaultLevel常量表示。默认的输出端为defaultOutput，并有用于存放所有Fabric模块日志的级别映射modules map[string]string，从类型上看其存储的日志级别都字符串化了。最后还有一个存放每个peer启动开始时的日志级别的映射peerStartModules map[string]string，由在每个peer启动完成之时调用SetPeerStartupModulesMap()初始化，并可通过调用RevertToPeerStartupLevels()恢复初始值。

init()函数通过调用Reset()函数等初始化一系列默认值，如默认的日志输出级别被设置成标准错误级别，不要设置成info级别。最后调用initgrpclogger()初始化grpc的日志对象。

在许多各级不同模块的源码中，在全局的开始处都有一句类似的调用，如在fabric/peer/main.go中为var logger = flogging.MustGetLogger("main")，这就是调用MustGetLogger函数生成一个名为指定字的日志对象，用以记录该模块的日志，并用安全的方式（用锁的方式）将该对象记录日志的级别备案到modules中。MustGetLogger函数内部依然用的是go-logging库的相应函数logging.MustGetLogger()生成的日志对象。

在logging.go中的其他函数基本都封装go-logging库函数，供fabric全局使用。如SetModuleLevel函数，常被调用，其实就是封装了go-logging库中的logging.SetLevel()函数，以达到符合Fabric使用要求的目的。可谓万变不离其宗。

5.4 配置系统

Fabric的配置系统是程序原始数据的来源之一，虽然简单却很重要。在阅读源码过程中对于具象化程序也很有帮助。我们将目光聚焦在/fabric/peer/main.go的main函数上，除了一系列mainCmd的命令操作，还有viper进行的一系列配置操作，并通过err := common.InitConfig(cmdRoot)进行了配置的初始化。

Fabric索取配置的途径有：环境变量、命令行参数、各种格式的配置文件。其中以配置文件为主，环境变量和命令行参数辅助，三者可以相互作用。主要的配置文件有core.yaml、orderer.yaml等，在/fabric/sampleconfig中有示例。主要使用的配置代码集中在/fabric/core/config下。

5.4.1 viper简介

Fabric的配置系统主要运用第三方包viper，可在github.com/spf13/viper下载。viper可以对系统环境变量、yaml/json等格式的配置文件甚至是远程配置进行读取和设置，并可以在不重启服务的情况下动态设置新的配置项的值并使之实时生效，是一个专门处理配置的解决方案。而且，viper（眼镜蛇）与cobra是伙伴，足见使用viper的重要性。

viper的基础用法如下：

```
//设置一个要读取的配置文件名（不包含后缀），一个viper只支持一个文件名
viper.SetConfigName("config")
//设置一个搜索配置文件的路径，viper的搜索路径可以有多个
viper.AddConfigPath("/etc/appname/")
viper.AddConfigPath(".")
//读取配置文件
viper.ReadInConfig()
//获取其中一个name项的值
viper.Get("name")
//将name的值设置为Bill
viper.Set("name", "Bill")
```

5.4.2 viper 搜索路径和文件

peer 命令对 core.yaml 的引入也是通过 viper，具体过程如下：

- /fabric/peer/main.go 中定义 const cmdRoot = "core"。
- main 函数中调用 err := common.InitConfig(cmdRoot)，该参数一路向下传递。
- InitConfig 函数在 /fabric/peer/common/common.go 中定义，其中调用了 config.InitViper (nil, cmdRoot) 和 viper.ReadInConfig()。
- InitViper 在 /fabric/core/config/config.go 中定义，接收 cmdRoot 作为参数，最终调用了 viper.SetConfigName()，将 core 设置为了配置文件名。
- common.InitConfig(cmdRoot) 中的 viper.ReadInConfig() 则读取了该配置文件。

orderer 命令则使用 orderer.yaml 配置文件，由 viper 引入，具体过程如下：

- 在 /fabric/orderer/main.go 中 main 函数调用了 config.Load()。
- Load 在 /fabric/orderer/localconfig/config.go 中定义。该文件中定义了 Prefix = "ORDERER" 和 configName string，并在 init 初始化函数中将 configName 赋值为 strings.ToLower (Prefix)，即 orderer（所用的配置文件名）。Load 函数新建了一个专用于 orderer 的 viper，并调用了 cf.InitViper(config, configName)，其中 config 参数为新建的专用于 orderer 的 viper，configName 为配置文件名 orderer。
- InitViper 在 /fabric/core/config/config.go 中定义，最终调用了 viper.SetConfigName()，将 orderer 设置为了配置文件名。
- Load 随后调用了 config.ReadInConfig()，读取了配置文件。

5.4.3 InitViper

上述步骤中，peer 和 orderer 在初始化配置文件时，最终都将调用的终点指向了 /fabric/core/config/config.go 中定义 InitViper()。下面集中分析 InitViper。

1）判断环境变量 FABRIC_CFG_PATH 是否有值，如果有值，则是手工定义了 fabric 的配置文件所在路径。参考 Getting Started 中关于手工设置 export FABRIC_CFG_PATH=$PWD（当前目录）。

2）若没有定义该环境变量的值，则用代码添加 3 个路径作为搜索配置文件的路径：当前工作目录，$GOPATH/src/github.com/hyperledger/fabric/sampleconfig，/etc/hyperledger/fabric。

调用 SetConfigName() 设置配置文件名，所指的配置文件名 configName 是由参数传递的。

经由 InitViper，形成了以下 viper 配置：

搜索路径（二选一）

- FABRIC_CFG_PATH 指定的路径
- /，$GOPATH/src/github.com/hyperedger/fabric/sampleconfig,/etc/hyperledger/fabric

搜索的配置文件名

- core——核心配置，供各个模块使用
- orderer——orderer 配置，orderer 使用

另外，注意 InitViper 的第一个参数 v *viper.Viper。在 InitViper 函数中，无论是添加搜索路径（使用的是 addConfigPath 函数），还是设置要搜索的配置文件名（viper 自身的 SetConfigName 函数），都分为全局的 viper 和特定的 viper（也就是参数 v）。最终由 viper.AddConfigPath 或 viper.SetConfigName 完成的是全局的，由 v.AddConfigPath 或 v.SetConfigName 完成的，则是特定的。这样就可以很方便地初始化需要单独使用 viper 的模块，如 orderer 就是单独使用一条毒蛇（viper），其在 /fabric/orderer/localconfig/config.go 中的 Load 函数中，config := viper.New() 新养了一条自己的蛇（viper），然后将此蛇通过参数传给 InitViper、cf.InitViper(config,configName)。

5.4.4　安全文件配置

安全配置相关的代码在 /fabric/peer/main.go 中没有体现，而是在 /fabric/peer/node/start.go 中的 serve 函数中才初次出现。若 grpc 服务中使用了 TLS 网络，则需要 .key、.crt、.ca 配套文件。在此简略介绍，关于 TLS，将会在专门的章节中详细介绍。

在 /fabric/peer/node/start.go 中 的 serve 函 数 中，secureConfig, err := peer.GetSecureConfig()，获取安全配置。

使用的安全配置结构为 /fabric/core/comm/server.go 中定义的 SecureServerConfig，用于一个 grpc 服务端实例。

.key、.crt、.ca 文件所在的目录都是在 core.yaml 中定义的 tls 文件夹中，当使用 TLS 网络时，会读取这些文件的数据到 SecureServerConfig 对象中。在 GetSecureConfig() 函数中，使用 ioutil.ReadFile 读取在 /fabric/core/config/config.go 中定义的 config.GetPath(“…”) 函数，获取的 tls 路径下的相应文件，如 /etc/hyperledger/fabric/tls/server.crt。

5.4.5　命令选项配置

以 peer start 命令为例，在 /fabric/peer/node/start.go 中 startCmd() 函数中，flags.BoolVarP

(&chaincodeDevMode, "peer-chaincodedev", "", false,"Whether peer in chaincode development mode") 设置了 peer start 命令的选项之一为 peer-chaincodedev，用于赋值文件中的全局变量 chaincodeDevMode，该变量指定了 chaincode 的模式。chaincode 的模式在 core.yaml 中也有定义，chaincode.mode 的值为 net，为默认选项。而当执行 peer start 命令时，指定了选项 peer-chaincodedev=true，也即将 chaincodeDevMode 赋值为 true，在 serve() 函数中，就会使用 viper.Set("chaincode.mode", chaincode.DevModeUserRunsChaincode)，将 chaincode 的模式值设置成 dev。

5.4.6 环境变量配置

在 Fabric 目前的阶段，各个 peer 都是在容器中运行的，因此环境变量指的是各个容器中的环境变量。在各个容器的启动脚本中对容器的一些环境变量也进行了设置。在 peer-base-no-tls.yaml 中，对 peer 容器设置了如下环境变量：

```
- CORE_PEER_ADDRESSAUTODETECT=true
- CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
- CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=e2ecli_default
- CORE_LOGGING_LEVEL=ERROR
...
```

在 fabric/peer/main.go 中的开头，即获取容器的环境变量并设置：

```
//设置了环境变量前置，在此也就是peer
viper.SetEnvPrefix(cmdRoot)
//将环境变量加载进来了
viper.AutomaticEnv()
replacer := strings.NewReplacer(".", "_")
//将环境变量中的_换成.，这样就和yaml文件的配置相匹配了。
//因为viper读取yaml文件所形成的配置项就是按层级并以.分隔的格式，如peer.address
viper.SetEnvKeyReplacer(replacer)
```

5.5 账本

5.5.1 账本简介

ledger，中文就是账本、账册的意思。先来讨论 Fabric 账册的原因是 peer node start 所执行的函数 serve 中第一句代码即为 ledgermgmt.Initialize()，字面上看属于对账册进行初始化操作。换句话说，对账本相关环境的初始化，是其他 peer 命令执行的基础之一。

账本源码目录如下：

- common/ledger
- core/ledger

也就是说 Fabric 对于 ledger 的源码，把其他模块用得到的、有共享属性的部分放到了 common 目录下，核心的代码放到了 core 目录下。

Fabric 中的 ledger 实际就是一系列数据库存储操作。对应所选用的数据库，主要有两种：goleveldb 和 CouchDB。在 core.yaml 配置文件中的 ledger 区域中，stateDatabase 选项即为指定所选用的数据库，默认选用 goleveldb。本章着意于 ledger 的操作，因此下面也只以 goleveldb 为例进行介绍。

goleveldb：主要使用了第三方库 leveldb。下载地址 github.com/syndtr/goleveldb/leveldb。leveldb 是一个典型的 key-value 数据库，是谷歌公司做出来的，有多种语言版本，在此自然是使用 goleveldb。把 Fabric 的 ledger 称为 kviedger，很好地体现了账本的特性，数据的操作都是基于键 – 值。goleveldb 数据库定义、操作的代码集中在 common/ledger/util/leveldbhelper 目录下。

CouchDB：是 Fabric 用 go 语言手工实现的一个数据库，源码集中在 core/ledger/util/couchdb 下。目前 couchDB 在 Fabric 中只用于版本数据库所使用的两个方案中的一个。

Fabric 支持 leveldb 和 CouchDB 数据库对世界状态的持久化存储，此处对默认的 leveldb 的基本操作如下：

- 打开数据库

```
db, err:=leveldb.OpenFile("./db", nil)
```

作用就是在当前目录下创建一个 db 文件夹作为数据库的目录。

- 存储键值

```
db.Put([]byte("key1"),[]byte("value1"),nil)
```

作用就是在数据库中存储键值对 key1-value1。leveldb 数据库中对键值的操作都是 byte 格式化的数据。

- 获取键值对

```
data,_ := db.Get([]byte("key1"),nil)
```

获取 key1 对应的值。

- 遍历数据库

```
iter := db.NewIterator(nil, nil), for iter.Next(){ fmt.
Printf("key=%s,value=%s\n",iter.Key(),iter.Value()) }, iter.Release()
```

作用就是建立迭代器 iter，然后依次遍历数据库中所有的数据，并打印键和值，最后释放迭代器 iter。

- 关闭数据库

```
db.Close()
```

Fabric 中到处都是接口，各个层级的代码编写风格和习惯很一致，甚至使用的函数名、对象名都有大量雷同的，因此有阅读源代码显得十分费神，其实阅读源代码时应该遵循以下

两条原则：

1）无论（概念上或形式上）多么复杂的对象，其本质也不过是一个结构体和挂载到该结构体上的一些操作函数而已。

2）无论对象的初始化多么复杂，其本质也不过是声明后填充该对象中的各个字段的过程而已。挂载到该对象的函数无论多么复杂，也不过是对该对象中的成员所承载的数据进行增、删、改、查操作而已。

因此，追溯并查看 ledgermgmt.Initialize() 函数，在 core/ledger/ledgermgmt/ledger_mgmt.go 中，其直接 once.Do 了 initialize() 函数。在 initialize() 函数中，除了日志操作和锁保护（一般在初始化的函数中都会进行锁保护，可参看其他源码）外，所做的只是初始化了 3 个对象：openedLedgers、ledgerProvider 和 initialized，这 3 个对象均为文件中的全局变量。

```
initialized = true
openedLedgers = make(map[string]ledger.PeerLedger)
provider, err := kvledger.NewProvider()
ledgerProvider = provider
```

其中 initialized 和 openedLedgers 分别赋了初值和分配了内存，openedLedgers 是存放 peer 的账本映射，initialized 则为是否初始化的标识，两者并未有进一步操作，可暂时搁置一旁。而 ledgerProvider 则被赋予由 kvledger.NewProvider() 函数返回的值，从其名字我们就可以知道该对象是一个账本服务提供者，因此我们的目光停留在此就可以了。其原型为 ledger.PeerLedgerProvider 接口，在 core/ledger/ledger_interface.go 中定义如下：

```
type PeerLedgerProvider interface {
    Create(genesisBlock *common.Block) (PeerLedger, error)
    Open(ledgerID string) (PeerLedger, error)
    Exists(ledgerID string) (bool, error)
    List() ([]string, error)
    Close()
}
```

根据 Fabric 的惯例，有 Provider 字样的对象，不管大小，都是某一主题模块服务的提供者，提供该主题模块的一系列操作服务。而接口类型的 Provider 对象则在具体实现上会分为多种具体的 Provider 以供使用（同时也留下了扩展空间）。ledger 的 Provider 也如此，kvledger.NewProvider() 函数返回的就是 PeerLedgerProvider 接口的一个具体实现，kvledger provider，即键值账本提供者，在 core/ledger/kvledger/kv_ledger_provider.go 中定义如下：

```
type Provider struct {
    idStore           *idStore                      //ledgerID数据库
    blockStoreProvider blkstorage.BlockStoreProvider //block数据库存储服务对象
    vdbProvider       statedb.VersionedDBProvider   //状态数据库存储服务对象
    historydbProvider  historydb.HistoryDBProvider  //历史数据库存储服务对象
}
```

还是根据 Fabric 的惯例，在每个定义对象结构的文件里，通常都会有一个专门用于生成该对象的函数，kvledger.NewProvider() 就是用于生成键值账本服务提供者的函数。通

过追溯，我们会发现，这个对象中的 4 个成员对象其实就是 4 个数据库，分别用于存储不同的数据，也是账册存储所需要的。而 kvledger.NewProvider() 所做的就是分别按照配置生成这 4 个数据库对象，这也符合前面我们所说的两个原则。4 个数据库中，除了 idStore 和 blockStoreProvider 有自己特殊的配置外，其余都共同使用 leveldb 数据库存储服务提供者。

5.5.2　数据存储服务对象

以 block 数据库存储服务对象 blockStoreProvider 的结构为例，其代码集中在 commom/ledger/blkstorage 下（本节中除介绍 leveldb 数据库存储服务对象外，所涉及路径皆以此路径为基准）。blockStoreProvider 的原型 blockStoreProvider 依然是个接口，在 blockstorage.go 中定义此接口，具体实现为用文件系统存储，即 fsblkstorage/fs_blockstore_provider.go 中定义的 FsBlockstoreProvider。

也就是说，与块数据存储服务对象 blockStoreProvider 最终对接的是 3 个成员，其中两个配置项成员 conf 和 indexConfig 是相较于其他数据库服务对象所独有的，一个 leveldb 数据库存储服务提供者 leveldbProvider 则和其他数据库服务对象一样。而专门用于初始化 FsBlockstoreProvider 的函数即为 fsblkstorage.NewProvider()。

在 kvledger.NewProvider() 中，以下 3 句代码用于初始化 blockStoreProvider 对象。

```
attrsToIndex := []blkstorage.IndexableAttr{
    blkstorage.IndexableAttrBlockHash,
    blkstorage.IndexableAttrBlockNum,
    blkstorage.IndexableAttrTxID,
    blkstorage.IndexableAttrBlockNumTranNum,
    blkstorage.IndexableAttrBlockTxID,
    blkstorage.IndexableAttrTxValidationCode,
}
indexConfig := &blkstorage.IndexConfig{AttrsToIndex: attrsToIndex}
//传递两个配置项，并在内部使用专用函数生成一个leveldb数据库对象，最终生成块数据存储服务对象
blockStoreProvider := fsblkstorage.NewProvider(
    fsblkstorage.NewConf(ledgerconfig.GetBlockStorePath(),
                         ledgerconfig.GetMaxBlockfileSize()),
    indexConfig)
```

块索引配置 indexConfig 用于存储块索引字段值，这在 blockstorage.go 中进行了硬编码。可以将其想象成数据库表中准备为哪些字段建立索引，因而在此记录一下。

块存储配置对象在 fsblkstorage/config 中定义，blockStorageDir 和 maxBlockfileSize 两个字段指定了块数据库存储服务对象所使用的路径和存储文件的大小。在 fsblkstorage.NewProvider() 中，传入的 config 是用 NewConf(ledgerconfig.GetBlockStorePath(),ledgerconfig.GetMaxBlockfileSize()) 进行创建的，而 NewConf 又使用了 ledgerconfig 下的函数，分别获取了路径和大小。而通过追溯 ledgerconfig，发现其最终形成的路径值为 /var/hyperledger/

production/ledgersData/chains，大小为 64MB。

leveldb 数据库存储服务对象为 blockStoreProvider，它是实际最终操作数据库数据的对象，账本所使用的 4 个数据库服务对象均使用此数据库对象对数据进行操作。在 common/ledger/util/leveldbhelper/leveldb_provider.go 中定义如下：

```
type Provider struct {
    db        *DB
    dbHandles map[string]*DBHandle
    mux       sync.Mutex
}
```

leveldb 数据库存储服务对象包含了封装 leveldb 数据库对象的 db，一个数据库映射 dbHandles 和一把保护锁 mux。因为数据库是对数据的读写操作，所以有一把保护锁很正常，也很有必要。其处于同一个文件中专用的初始化函数 leveldbhelper.NewProvider() 中，在 fsblkstorage.NewProvider() 函数中即有体现：

```
p:= leveldbhelper.NewProvider(&leveldbhelper.Conf{DBPath: conf.getIndexDir()})
```

在此仍需关注的是 DB 结构中自带一个 leveldbhelper.Conf 配置选项，定义了 leveldb 数据库所在的目录。在 leveldbhelper.NewProvider() 初始化的过程中，使用了上层对象——块数据库存储服务对象中的 conf 所挂载的函数 getIndexDir()，在 common/ledger/blkstorage/fsblkstorage/config.go 中定义，获取了一个路径，最终被初始化在 chains/index 下。

Fabric 账本服务对象的目录结构如下：

回到 kvledger.NewProvider() 函数中，其他几个数据库的初始化过程和块数据库存储服务对象与之类似，但更简单一些，基本上都只是用专用函数初始化了一个 leveldb 数据库存储服务对象。至此，整个账本服务对象初始化完毕。以下列出账本服务对象的目录结构，具象化一下。

- /var/hyperledger/production(core.yaml 定义的 flieSystemPath 的值)
  - ledgersData// 账本目录
    - ledgerProvider//ledgerID 数据库目录
    - chains//block 块存储数据库目录
      - index//biok 索引数据库目录
      - chains
        - 账本 ID1
        - 账本 ID2
        - …
    - stateLeveidb// 状态数据库目录
    - historyLeveidb// 历史数据库目录

在 kvledger.NewProvider() 函数中接近结尾的地方，有一句代码需要注意：provider.rec

overUnderConstructionLedger()，该句调用了账本服务对象的一个函数，主要用于恢复处理一些之前账本初始化失败的操作，从中牵扯出了一大堆函数调用。但对这些操作的理解需要建立在理解账本操作的基础之上。

5.5.3　四类账本

peer 使用的账本的直接对象是 core/ledger/kvledger/kv_ledger.go 中的 kvLedger 对象，包含四类数据库账本，分别是 idStore，BlakStore，VersionedDB，HistoryDB。除了 idStore 被更高层的 PeerLedgerProvider 管理外，其余三个账本均由 kvLedger 持有管理。

1. idStore

idStore 相对简单，对象为 core/ledger/kvledger/kv_ledger_provider.go 中的 idStore，直接使用了 leveldb。主要功能是存储创建的账本的 ID、使用一个在建标识（ConstructionFlag）来标记账本是否正确建立，类似于一个检查点。

（1）存储账本 ID

存储账本 ID 相当简单，只有增、查操作。增，以 ledgerKeyPrefix+ 账本 ID 为 key，以 gensisblock 为 value，组成键值对进行存储。查，有 ledgerIDExists、getAllLedgerIds 两个接口，前者判断一个账本是否存在，后者获取所有建立成功的账本 ID。

（2）ConstructionFlag

建立一个账本需要一定的时间，而若在建立账本过程中系统崩溃，则会出现账本部分建立的情况，leveldb 中可能残留部分账本数据，磁盘中可能残留部分账本文件，这些需要清除或修复，否则可能会影响账本的再次建立（实际上不影响，但是可以做，具体参看 kv_ledger_provider.go 中 的 runCleanup(...)）。idStore 中 以 underConstructionLedgerKey 为 key，账本 ID 为 value，组成键值对进行存储，从而作为一个在建标识，标记这个账本正在创建当中。在 kv_ledger_provider.go 中：

1）Create(genesisBlock) 创建账本时，在检查完该账本是否存在后，若账本不存在，则立即通过 provider.idStore.setUnderConstructionFlag(ledgerID) 向 idStore 中添加在建标识，对当前创建的账本进行标记。

2）provider.openInternal(ledgerID)，ledger.Commit(genesisBlock)，先创建账本，这需要一系列的操作，然后再向账本中存储 genesisBlock。

3）provider.idStore.createLedgerID(ledgerID, genesisBlock)，在这个函数中，batch.Put+(key,val) 向 idStore 中存储该账本 ID，batch.Delete(underConstructionLedgerKey) 从 idStore 中删除在建标识。最后 s.db.WriteBatch(batch,true) 同步地批量写入这两点改变，如此这般这个账本才算完整地建立起来。而在 WriteBatch 将在建标识真正从 idStore 中删除前任何时候系统发生崩溃，下次系统重启读取 idStore 时，在建标识仍能被读取出来，因此在建标识可以标识账本部分在建的情况，此时需要执行 recoverUnderConstructionLedger 来恢复账本。

（3）账本恢复

ConstructionFlag 标志标识最新的账本是否成功建立。这里分为两种情况，即 gensisBlock 成功写入与否。在创建账本管理对象 PeerLedgerProvider 时，会主动尝试恢复一下账本（参看 kv_ledger_provider.go 中的 NewProvider()）。在 recoverUnderConstructionLedger() 中：

1）ledgerID, err := provider.idStore.getUnderConstructionFlag() 获取当前 idStore 中的在建标识，若不存在，则 ledgerID == "" 成立，说明当前不存在不完整的在建账本，程序返回。否则会继续执行，实际进行账本的恢复工作。

2）ledger, err := provider.openInternal(ledgerID)，再次创建 VersionedDB，BlockStore，HistoryDB。bcInfo, err := ledger.GetBlockchainInfo()，从 BlockStore 中读取 block 账本信息（包括当前账本高度，当前账本最新存储的 block 的哈希值，上一块 block 的哈希值）。

3）switch bcInfo.Height { ... }，根据获取的账本高度，分成两种情况：case 0 说明 gensisblock 未完整写入，则执行 provider.runCleanup(ledgerID)，provider.idStore.unsetUnderConstructionFlag()，即清理后删除在建标识，说明两点，一是 block 序列号确实是从 0 开始的，二是在未写入 gensisblock 的情况下账本算作未建立；case 1 说明 gensisblock 已写入，则执行 genesisBlock, err := ledger.GetBlockByNumber(0)，provider.idStore.createLedgerID(ledgerID, genesisBlock)，获取 gensisblock 块，并据此在 idStore 中添加这个账本。

这里需要注意的是：这里的账本恢复是在账本级别的层面上进行的，而每个账本在创建时进行的各种操作因系统崩溃而造成的部分写入的情况，都由各自账本自身负责恢复或清除（这将在讲其他账本时详述）。而且在此基础上，恢复工作的重点是修复和同步数据，而不在于创建直接可以使用的账本对象（创建可以使用的账本对象由其他接口负责，如 kv_ledger_provider.go 中的 Create(genesisBlock)，Open(ledgerID)），因此第 2 点所新创建的各个账本对象，只是为了让这些账本做一下自己所负责的恢复工作而已，这一点可以从第 2 点 bcInfo, err := ledger.GetBlockchainInfo() 之后就执行了 ledger.Close()（将创建的账本对象关闭）的操作看出来。

2. BlockStore

BlockStore 是一个底层的、基础的、公用的、存储 block 块数据对象的接口，具体实现为 common/ledger/blkstorage/fsblkstorage/blockfile_mgr.go 中的 blockfileMgr，统筹管理 block 的存储操作。block 的存储状态信息和 block 数据自身是分开存储的：当前 block 块的存储状态存储在 leveldb 数据库中，block 块数据自身则存储在称为 blockfile 的文本文件中。换句话说，leveldb 用于存储当前账本的 block 数据的保存状态信息，这些信息主要指 block 的位置、长度信息、当前 blockfile 文件大小、index 信息等，主要是用来确定 block 在 blockfile 中的所在位置，而 blockfile 文件保存实际的 block 数据。基本的操作方式也是根据 leveldb 中的信息去 blockfile 中对 block 进行读写操作，两者相互配合，共同保证

block 存储的完整性。另外有一个 block 迭代器对象，用于逐个读取 block 块数据的对象。

这里涉及几个对象（以下所说的 block 均指账本中当前存在的最新的 block，目录以 common/ledger/blkstorage/fsblkstorage/ 为基准）。

（1）blockfile

它是存储 block 块数据的文件。具体实现为 block_stream.go 中的 blockStream，主要对 blockfile 进行读写操作。blockfile 文件名以 blockfile 为前缀，以 6 位数字为后缀，依次增加，如 blockfile_000000，blockfile_000001，…。

首先说一下 block 数据在 blockfile 中具体的存储方式，存储方式决定了如何读取和写入 block。一个完整的 block 数据包包含两部分：block 数据长度信息和 block 数据自身，这里分别用 A 和 B 表示，block 数据包在 blockfile 中的偏移也是以 A 为开始算起的（该例子后面依旧使用）。如一块 block 数据为 blockBytes，则其长度是一个 uint64 类型的整数，为 blockBytesLen = uint64(len(blockBytes))，为了节约空间和方便读取，会用 blockBytesEncodedLen := proto.EncodeVarint(blockBytesLen) 对这个整数编码，形成一个变长的数据（原理是在大端系统中删除一个整数低位部分的 0）。这里的 blockBytesEncodedLen 就是 A，blockBytes 就是 B。然后将 A 和 B 先后写入 blockfile，这块 block 数据就写入完成。注意，A 是代表了 B 的长度的数据，A 本身也有个长度，所以 len(A)+len(B) 才是这个 block 数据包的长度。其他的对象，比如保存点 checkpointInfo，会在写入的时候记录下该块 block 在 blockfile 中的位置等信息。

从 blockfile 中读取 B 时，会直接在 block 数据包的开始处（也就是从 A 开始）直接读取 8 字节（A 所在的位置由 leveldb 存储的 checkpointInfo 保存），然后调用 length, n := proto.DecodeVarint(lenBytes) 尝试解码 A 的实际值。这里直接读取 8 字节，是假设这 8 字节中一定会完整包含 A。笔者试验了一下，对一个 512 万亿的数字进行解码（EncodeVarint），得到的数据的长度也只有 7 位，也就是说，这里假设一个 block 的值被解码（EncodeVarint）后的数据一定小于或等于 8 位。另外，如 A 只有 3 位，读取的 8 位数据的剩余 5 位为 B 的数据时，DecodeVarint 依然会解码出 A 来，此时返回的 n 为 3，length 则为 A 代表的实际值（也就是 B 的实际值）。然后，根据解码出来的 length，以及 A 所占的 n 字节，则可以定位到 B 开始的地方，读取 length 个字节，也就完整地把 B 读取出来了。对于 B 是否完整，是通过比较当前 blockfile 与 length+n+checkpointInfo 记录的 block 数据包的偏移值之和的值来判定的，若后者小于当前 blockfile 的实际值，则自然是完整的。

读取 block 数据包，是由 block_stream.go 中的 blockStream 和 blockfileStream 对象完成的。其中能体现上述读取过程的是两个对象的 nextBlockBytesAndPlacementInfo 接口。该接口每次从指定的文件中的指定位置开始，尝试读取一个 block 数据包。

（2）block 信息数据库

它是存储当前账本中 block 存储状态信息的 leveldb 数据库，具体到 blockfileMgr 对象中的 db、index 两个成员（两者实际上使用的是同一个 leveldb 数据库）。它主要存储两种数据：

1）检查点 checkpoint，具体为 blockfile_mgr.go 中的 checkpointInfo 对象。每当在 blockfile 中存储一个 block 数据包，都会在 leveldb 中存储该 block 数据包对应的检查点。检查点以 blkMgrInfoKey 为 key（blockfile_mgr.go 中），checkpointInfo 为 value，组成一个键值对存储在 leveldb 中。latestFileChunkSuffixNum 记录 block 所在的 blockfile 的文件名后缀，latestFileChunksize 记录当前 blockfile 最新的大小，isChainEmpty 标识当前账本是否为空，lastBlockNumber 记录账本当前最新 block 的序列号。

2）索引 index，具体为 blockindex.go 中的 blockIndex 对象。每当在 blockfile 中存储一个 block 数据包，都会在 leveldb 中存储该 block 数据包对应的一批索引。索引分别以 block 序列号、block 哈希值、交易 ID 等为 key，以账本中最新 block 的位置信息等为 value，组成一批键值对，存储在 leveldb 中。这些索引项的 key 值预定义在 common/ledger/blkstorage/blockstorage.go 中，主要是为调用者提供多种索引方式，在 blockfile 中定位 block 块数据。比如，用 block 的序列号查找一个 block 数据，或使用 block 数据的哈希值查找 block 数据，还可以使用交易 ID 查找一个 block 中的具体交易的数据等。block 数据包对应的每批索引中包含了 block 块中每笔具体交易数据的索引，具体存储在 blockIndex 对象的 txOffsets。这个成员在计算每笔交易在 blockfile 中的偏移位置时，随着添加 A+B，总共要经历三轮，首先计算的是每笔交易在 B 中的相对偏移，然后计算的是每笔交易在 A+B 中的相对偏移，最后计算的是每笔交易在 blockfile 中的偏移。索引自身也有一个检查点 indexcheckpoint，用于记录当前最新的 block 数据包的索引的存储情况。

在写 block 数据包时，当先后写入 A 和 B 后，进行如下两步：

1）创建一个新的 checkpointInfo 对象 newCPInfo，然后根据现有的 checkpointInfo 对象 currentCPInfo 和写入的 block 数据包填充 newCPInfo，再非同步地调用 db.Put(...)，将新的 checkpointInfo 写入 leveldb 中，覆盖掉 currentCPInfo。检查点的作用主要在于记录当前账本的保存状态，也就是说记录账本当前保存到哪个 block 了。

2）调用 indexBlock(...)，进而调用 batch.Put(...)，以批量写入的形式将各个索引对应的信息写入 leveldb 中。因为每批索引所使用的 key 不一样，因此不会发生覆盖。索引检查点会在每批索引的最后写入，也即当索引检查点存在，则相关索引一定存在。索引的作用主要是为调用者提供多种查找具体 block 或交易数据的方式。

在此总结一下，写入一个 block，从前到后所涉及的要依次写入的每个数据如下：

1）block 长度信息；

2）block 块数据；

3）blkMgrInfoKey-checkpointInfo；

4）blockHashIdxKeyPrefix+block 哈希值 -block 数据包位置信息；

5）blockNumIdxKeyPrefix+blockID-block 数据包位置信息；

6）txIDIdxKeyPrefix+ 交易 ID- 交易数据位置信息（每个 block 中所含的所有交易）；

7）blockNumTranNumIdxKeyPrefix+blockID+ 交易 Seq- 交易数据位置信息（每个 block

中所含的所有交易）；

8）blockTxIDIdxKeyPrefix+ 交易 ID-block 数据包位置信息（每个 block 中所含的所有交易）；

9）txValidationResultIdxKeyPrefix+ 交易 ID- 交易验证结果（每个 block 中所含的所有交易）；

10）indexCheckpointKey-block 序列号。

上述列表可划分出两个范畴，3 个小整体：1）和 2）存储到 blockfile 文件中，属于一个范畴，是一个小整体；3）～ 9）存储在 leveldb 中，均为键值对，属于一个范畴；3）为检查点键值对，是一个小整体；4）～ 10）为索引键值对，10）是索引检查点，是一个小整体。blockID 指 block 的序列号，交易 ID 指交易的序列号，交易 Seq 指交易在 block 的 Data 数组中的下标序号。所有的 key 的…Prefix 前缀均在 blockindex.go 中定义。3）和 10）的每次更新均会覆盖先前值。

block 数据包或交易数据的位置信息是一个 blockindex.go 中的 fileLocPointer 对象，该对象中，fileSuffixNum 记录数据所在的 blockfile 文件后缀，offset 记录数据在 blockfile 文件中的偏移，也就是数据从哪个位置开始，bytesLength 记录数据的大小（这个字段 block 数据包未使用，因为其长度存放在 A 中，没必要使用这个字段。而交易数据使用了这个字段，来记录每个交易数据的大小）。

同时，正因为在写入一个 block 时涉及多个、多类数据，并先后写入，因此存在当系统崩溃时数据部分写入和 3 个小整体数据不同步的情况。在重新启动系统建立新的 blockfileMgr 时，需要对部分写入的数据进行修补，并对 3 个小整体的数据进行同步。同步时，遵循的一个隐含的规则是前面的数据未写入，则后边的数据肯定没有写入，存在的情况有：① 1 未完整写入；② 1 完整写入，2 未完整写入；③ 1 ～ 2 完整写入；3 未完整写入；④ 1 ～ 3 完整写入，4 ～ 10 未批量写入。据此，同步的步骤如下：

1）从 leveldb 中获取存有的检查点信息 cpInfo（如果没有则说明是新建的账本）。

2）根据 cpInfo 中存储的 block 数据包的位置信息，定位该 block 数据包在哪个 blockfile 中的哪个位置。然后以此为开始尝试读取一个完整的 block 数据包，对 cpInfo 进行更新同步。

3）这里需要辨别的是前①～③种情况，这 3 种情况下，取出的 cpInfo 其实是上一个 block 数据包对应的 cpInfo，定位读取的时候也会完整读取出上一个 block 数据包，然后再读取到此 block 时，当是前两种情况时，由于当前 block 数据包数据未完整写入，因此会直接从上一个 block 数据包的末尾处截断 blockfile 文件，删除当前 block 数据包不完整的数据，此时也不需要再更新 cpInfo ；当是第③种情况时，由于当前 block 数据包数据已完整写入，因此会再读取当前 block 数据包，依据当前 block 数据包的信息，更新 cpInfo 和索引（索引会使用更新过后的 cpInfo 进行更新）。当是第④种情况时，取出的 cpInfo 为当前 block 数据包对应的 cpInfo，cpInfo 不用更新，而只需要对索引信息进行同步。索引同步时，会读取索引

仅供非商业用途或交流学习使用

的检查点信息，该索引检查点信息必然是上一个 block 数据包的索引检查点，而由此获取的 block 数据包位置信息也是上一块的 block，跳过该 block 后，就同 cpInfo 更新的方式类似。

账本操作包括创建 - 增 - 查。这里在代码中追溯 BlockStore 的主要操作，以下涉及的代码的基准目录为 common/ledger/blkstorage/fsblkstorage/，涉及的函数若未表明所在位置，请自行搜索。

（1）创建

1）在 blockfile_mgr.go 中，newBlockfileMgr(...) 创建了一个新的 block 存储管理者，接收了配置、索引配置和一个 leveldb 数据库 3 个参数。其中配置中包含了 block 存储路径等信息。leveldb 以参数的形式传入，给了更高层的管理对象更大的灵活性，如 fs_blockstore.go 中的 fsBlockStore，fs_blockstore_provider.go 中的 FsBlockstoreProvider。

2）cpInfo, err := mgr.loadCurrentInfo()，即从 leveldb 中取出最新的检查点数据。当 if cpInfo == nil，说明当前新建的 block 存储管理者是管理的一个比较新的账本，要么什么数据都未写入，要么写入了第一块 block 数据包的部分数据（参看上文数据部分写入的情况①～③），则可以将 cpInfo 更新为 blockfile 文件开始处（以供后续步骤进行尝试读取现有的 block 数据，然后更新 cpInfo）。

3）syncCPInfoFromFS(rootDir, cpInfo)，即根据取出来的 cpInfo 同步更新的函数。在这个函数中，scanForLastCompleteBlock(...) 是尝试读取最后 blockfile 中最后一块 block 的函数，所给的 3 个参数分别是哪个目录、哪个文件（后缀）、从文件里的哪个位置开始读，第 3 个参数赋的值为 int64(cpInfo.latestFileChunksize)，即 cpInfo 所对应的 block 数据包写入后文件的大小，也就是从 cpInfo 所代表的 block 数据包末尾的下一个位置（即下一个 block 数据包开始的位置）开始尝试读取一个完整的 block 数据包，如此，至多读取出一个完整的 block。读取的方法依然如上面所述，具体的实现是 block_stream.go 中的 nextBlockBytesAndPlacementInfo()。若读取了这个至多的一个新的完整的 block 数据包，则会更新 cpInfo 的 latestFileChunksize（指向 blockfile 中最新的 block 数据包尾）、lastBlockNumber（更新为最新 block 的序列号）、isChainEmpty（更新为账本非空）这 3 个字段。

4）currentFileWriter.truncateFile(cpInfo.latestFileChunksize)，根据更新过后的 cpInfo 所存储的最后一个 block 数据包的末尾位置，直接截断该 blockfile 文件。这比较好理解，更新过后的 cpInfo 的 latestFileChunksize 值之后的数据肯定是不完整的，所以直接截断，清除不完整的数据。

5）mgr.syncIndex()，根据更新过后的 cpInfo 和自身的检查点信息，创建同步索引。在这个函数中，lastBlockIndexed, err = mgr.index.getLastBlockIndexed() 即为获取索引的检查点，获取的是解码后的 block 序列号，即表示序列号为 lastBlockIndexed 的 block 数据包的索引已经完整存储。同样，若未获取，则说明所处理的账本较新，还没有索引存储到 leveldb 中。flp, err = mgr.index.getBlockLocByBlockNum(lastBlockIndexed) 是使用 blockNumIdxKeyPrefix+blockID 的索引方式获取序号为 lastBlockIndexed 的 block 数据包位置信息 flp。又因为能成功

仅供非商业用途或交流学习使用

获取索引的检查点，所以可知该 block 数据包一定是完整存储的，因此可以直接跳过这个 block 数据包而去尝试读取下一个 block，这也是函数中 skipFirstBlock 变量所起的作用。而一旦能成功读取下一个 block 数据包，则会调用 info, err := extractSerializedBlockInfo(blockBytes) 根据读取出来的 block 数据包创建新的索引信息，然后调用 mgr.index.indexBlock(blockIdxInfo) 将新的 block 数据包的索引写入 leveldb 中，从而完成索引的更新同步。

（2）增

1）addBlock(block) 对一个 block 数据块进行了添加。在这个函数中，if block.Header.Number != mgr.getBlockchainInfo().Height 首先验证了添加 block 数据的序列号是否是下一个 block 应有的，即序列号是依次连续的。blockBytes, info, err := serializeBlock(block) 将 block 数据串行化成可以直接写入 blockfile 文件中的数据，这里的 blockBytes 即为 B，也在这里第一次计算了每笔交易数据在 block 块内的偏移位置（即以 block.Data 为开始每笔交易的偏移）。currentOffset := mgr.cpInfo.latestFileChunksize 即为当前 blockfile 的大小，也就是准备开始写当前添加的 block 数据包的位置。blockBytesLen := len(blockBytes)，blockBytesEncodedLen := proto.EncodeVarint(uint64(blockBytesLen)) 压缩了 block 的值，这里的 blockBytesEncodedLen 即为 A。totalBytesToAppend := blockBytesLen+len(blockBytesEncodedLen) 则为 block 数据包的总大小。如果 if currentOffset+totalBytesToAppend>mgr.conf.maxBlockfileSize，即当前文件的大小 + 要添加的 block 数据包的大小之和大于配置中规定的 blockfile 的大小限制，则会调用 mgr.moveToNextFile() 直接新启用下一个 blockfile 文件来存储这个新的 block 数据包。

2）mgr.currentFileWriter.append(blockBytesEncodedLen, false)，非同步地在第 1 步确定的 blockfile 文件和文件中的位置处的写入 A，第 2 个参数给的是 false，因此此时 A 只是在缓存中而已。mgr.currentFileWriter.append(blockBytes, true)，第 2 个参数是 true，同步写入 B，即连同 A 一同实际地写入 blockfile 磁盘文件中。需要注意的是，虽然这里 A 和 B 是一起写入的，但只是为了减少磁盘文件的操作，第 2 个参数为 true，在 append(...) 函数（blockfile_rw.go 中）中会手动调用 file.Sync()，即手工刷新缓存将缓存中的数据写入实际的磁盘文件。但这个操作也不是事务性的，即在系统层面也是一字符一字符向磁盘文件中写的，因此如果在写的过程中发生系统崩溃，仍会出现 A 或 B 部分写入的情况。如果写入 A 或 B 的过程中出错，则 mgr.currentFileWriter.truncateFile(mgr.cpInfo.latestFileChunksize) 直接把文件从最开始写的地方截断，将可能新写入的部分 block 数据包的数据删去，然后添加程序结束并返回一个错误。

3）newCPInfo := &checkpointInfo{...}，根据写入的 block 数据包和现有检查点的信息，创建一个新的检查点。然后 mgr.saveCurrentInfo(newCPInfo, false) 非同步地将新的检查点写入 leveldb 数据库中。同样的，若是检查点添加错误，也是直接截断文件，删除写入的 block 数据包，将 blockfile 恢复如初，结束添加并返回一个错误。

仅供非商业用途或交流学习使用

4）blockFlP := &fileLocPointer{...}，根据新的检查点 newCPInfo 和开始写入 block 数据包时的偏移信息，创建一个 block 数据包的位置信息 blockFlP。for ... {txOffset.loc.offset += len(blockBytesEncodedLen)}，这里第 2 次计算 block 中包含的每笔交易的偏移，即加上了 A 所占的字节数，计算每个交易从 A 处开始算起的偏移量，也可以说是每笔交易在 block 数据包内的偏移。mgr.index.indexBlock(...)，将收集的索引使用到的关于新的 block 数据包的信息传入，创建新的一个 block 数据包的索引，在这个函数中，会逐个写入 1 ～ 6 的索引项。其中在添加索引 3 和 4（上面的数据 6 和 7）这两个与交易有关的索引时，会调用 txFlp := newFileLocationPointer(...)，第 3 次计算每个交易的偏移信息，即确定每笔交易在 blockfile 文件内的偏移，同时也会在 txFlp 中记录每笔交易的大小。在所有索引项都批量写入后，然后 batch.Put(indexCheckpointKey, encodeBlockNum(blockIdxInfo.blockNum)) 写入索引的检查点，最后 index.db.WriteBatch(batch, false) 非同步地写入 leveldb 数据库中。这里和第 3 点写入 leveldb 时都用了非同步写入，即执行后可能这些数据写入 leveldb 的缓存中而非数据库中，这样可以提高写入的效率。

5）mgr.updateCheckpoint(newCPInfo)，这个主要更新了 blockfileMgr 对象保存的检查点 cpInfo，然后调用了一个 mgr.cpInfoCond.Broadcast() 函数，这个函数主要使用 sync.Cond 的特性，服务于通知 block 迭代器的 Next() 可能存在的等待，让其继续执行。上面已经说过，迭代器的 Next() 在遍历到序列号大于账本当前已有的最新 block 数据包的序列号时，会进入等待，一直到新的 block 数据包成功存储到 blockfile 中为止，所以这里就是在成功添加了一个 block 数据包之后进行的通知。

6）mgr.updateBlockchainInfo(blockHash, block)，这个函数使用 atomic.Value 的特性，原子性的存储链（也就是账本）的信息，即一个 BlockchainInfo 对象，该对象 Height 保存了链的高度，CurrentBlockHash 保存了当前链存储的最新 block 的哈希值，PreviousBlockHash 保存了上一个 block 的哈希值。

（3）查

关于查询 block 数据很简单，不同索引提供了不同的查询方式，均集中在 blockfile_mgr.go 中，是一系列 retrieve__By__ 格式的函数。名字很好理解，第 1 个空是你要查的东西，第 2 个空是你根据什么查这个东西，即通过什么来检索什么，也即 By 什么来 retrieve 什么。这些函数主要通过 leveldb 中保存的 block 数据包的索引来获取 block 数据。

3. VersionedDB

VersionedDB 账本可以叫作状态账本，这个状态（state）是官方文档中所提及的“世界状态”，world state，具体即为由每个交易数据最新生成的一个关于交易读写集的有效的键值对。该数据库中有几个重要的对象，如交易模拟器、查询器、验证器、读写集等。

VersionedDB 使用的 state 数据库有两个版本，即 leveldb 版本和 CouchDB 版本，这里只讲 leveldb 版本。两者的主要区别在于：leveldb 只支持基本的、以 key 为基础的查询，而 CouchDB 则支持更丰富的查询手段，即富查询（参看 core/ledger/util/CouchDB/CouchDB_

仅供非商业用途或交流学习使用

test.go 中的 TestRichQuery），对于调用者来说也更接近事务逻辑（如可以根据某账户的特征，如颜色、尺寸等进行查询，只需要预先定义好）。state 数据库也不是直接由账本使用的，而是被一个可称作交易管理者的对象 TxMgr 持有并管理，账本通过这个交易管理者向 state 数据库读写交易数据。同时，TxMgr 对象也提供其他功能，比如提供交易模拟器、交易查询器、交易验证器，这些“器”将在读写交易数据的时候发挥作用。TxMgr 的基础目录在 core/ledger/kvledger/txmgmt/ 下（该章节以此目录为基准），具体的实现为 txmgr/lockbasedtxmgr/lockbased_txmgr.go 中的 LockBasedTxMgr。

在 peer 节点中使用 VersionedDB 的过程：用户使用 peer 节点发起一笔交易，如 ACC 的部署，是一个明显的会写入数据的交易；再如用户执行 peer chaincode query ...，则是一个明显的会读出数据的交易。在交易过程中，这些读取和写入的交易数据会通过 TxSimulator（交易模拟器）放入一个叫作读写集的容器，在交易返回时，再使用 TxSimulator 统一读取，然后将读取的数据放入 ProposalResponse 返回。返回后读集中的数据直接打印，写集中的数据被放入 Envelope 中再发送给 orderer（请参看 peer/chaincode/common.go 中的 ChaincodeInvokeOrQuery）。orderer 节点依据 Envelope 生成 block 后，再返回给 peer 节点。peer 节点接收后，经 gossip 模块接收并提交到账本对象后，账本对象添加 block，会添加到 VersionedDB 账本中，VersionedDB 又会将 block 中交易的数据最终添加到 state 数据库中。以执行 chaincode_example02.go 的 invoke(stub,args)（该函数简单查询了 A、B 两个账户的余额，并由 A 向 B 转了一笔钱，即形成了 A、B 两个账户转账调整后各自新的余额值）为例，步骤如下：

1）交易发起端执行 peer chaincode invoke -n chaincode_example02 -c '{"Args":["invoke","a","b","10"]}'，通过 peer/chaincode/invoke.go 中的 chaincodeInvoke(...) 函数向 endorser 节点发送请求。假设该交易为 Tx_A。

2）core/endorser/endorser.go 中，ProcessProposal –> txsim, err = e.getTxSimulator(chainID)，创建了一个交易模拟器。

3）e.simulateProposal(...,txsim) –> e.callChaincode(...,txsim)。

4）经过一系列辗转，会在 core/chaincode/shim/handler.go 中，handleTransaction 中调用 res := handler.cc.Invoke(stub)（即 chaincode_example02.go 中的 Invoke）–> 根据第 1 步的参数，在 Invoke() 中进入 if function == "invoke" 分支，从而调用 invoke() –> 再经过一系列辗转，会在 core/chaincode/handler.go 中，enterBusyState，通过交易模拟器的 txsimulator.GetState()、txsimulator.SetState(...)，将 A、B 账户的数据放入读集，把更改的 A、B 账户的数据放入了写集（其他交易会用到了其他的 DeleteState()，GetStateRangeScanIterator()，ExecuteQuery()），这一步相当于模拟执行了交易。

5）重回 core/endorser/endorser.go 中，simResult, err = txsim.GetTxSimulationResults() –> 返回至 ProcessProposal()–> pResp, err = e.endorseProposal(...,simulationResult,...) –> e.callChaincode(...)，获取交易的写集，并进行了第二轮的 callChaincode，一系列调用后会进入下一步。

仅供非商业用途或交流学习使用

6）core/scc/escc/endorser_onevalidsignature.go 中，Invoke(...) -> presp, err := utils.CreateProposalResponse(...,results,...)，results 为交易 Tx_A 的结果集，被放入 ProposalResponse，返回至第 5 步的 endorseProposal，再返回给交易发起端。

7）peer/chaincode/common.go 中，ChaincodeInvokeOrQuery() -> env, err := putils.CreateSignedTx(...)，把返回的 ProposalResponse 打包成 Envelope -> err = bc.Send(env)，发送给 orderer -> …，orderer 处理 block 过程省略 -> orderer 将形成的 block 发送给 peer 节点，再经 gossip 模块散播使用 committer 模块的过程。

8）committer 向账本提交 block 时，最终定位到 core/ledger/kvledger/kv_ledger.go 中的 Commit(block) -> err = l.txtmgmt.ValidateAndPrepare(block, true)，l.txtmgmt.Commit() 即是使用交易管理者向 state 提交交易数据。ValidateAndPrepare 做两件事，一是使用验证器验证交易的读写集，以确定交易的有效性；二是若交易有效，则将交易的写集中的数据放入数据升级包中，为下一步的 l.txtmgmt.Commit() 提交这批数据做准备。

交易模拟器 TxSimulator 和交易查询器 QueryExecutor 的接口在 core/ledger/ledger_interface.go 中定义，具体实现为 txmgr/lockbasedtxmgr/ 下 lockbased_tx_simulator.go 中的 lockBasedTxSimulator 和 lockbased_query_executer.go 中的 lockBasedQueryExecutor。通过账本 kvLedger 的 NewTxSimulator()，NewQueryExecutor() 接口可获取 TxSimulator 和 NewQueryExecutor。两者均直接使用 helper.go 中的 queryHelper 实现了查询功能，这点从 queryHelper 这个名字可以看出来。TxSimulator 实际上包含 QueryExecutor，而 QueryExecutor 算是 TxSimulator 在查询功能上的增强和拓展（其中交易查询器的 ExecuteQuery 是富查询接口，只支持 CouchDB 版本的 VersionedDB），因此直接使用的一般是交易模拟器。除了查询，TxSimulator 还提供状态的写入功能，写入分为增加和删除。因此，TxSimulator 和 QueryExecutor 涉及的操作就有查、增、删。

参看 txmgr/lockbasedtxmgr/lockbased_tx_simulator.go 和 helper.go，交易模拟器的读、写、删的操作步如下：

1）GetState(ns, key) -> q.helper.getState(ns, key)，依据名字空间和 key，从 state 数据库中获取一个状态，返回并写入读集（返回的是值，写入读集的是值的版本）。这里的名字空间是 chaincodeID，也即对应一个 chaincode 的每一个交易使用一个读写集，下同。

2）SetState(ns, key, value) -> s.rwsetBuilder.AddToWriteSet(ns, key, value)，将一个值写入写集。

3）DeleteState(ns, key) -> SetState(ns, key, nil)，删除一个值，当给的 key 的 value 为 nil 时，即表示要将此 key 的值置为 nil，即删除这个值。

4）SetStateMultipleKeys(ns, kvs) -> for ... { SetState(...) }，一次性设置多个键值对，写入交易的写集。

5）GetStateMultipleKeys(...) -> q.helper.getStateMultipleKeys(...)，依据一个名字空间和一批 key，从 state 数据库中获取一批状态。

仅供非商业用途或交流学习使用

6）GetStateRangeScanIterator(...) -> q.helper.getStateRangeScanIterator(...)，依据一个命名空间、开始的 key、结束的 key，从 state 数据库中获取一个指定范围的交易查询迭代器。

7）ExecuteQuery(namespace,query) -> q.helper.executeQuery(namespace,query)，这个接口对于 leveldb 来说不支持。

8）Done() -> q.helper.done()，交易查询器执行完毕。

TxSimulator 之所以叫交易模拟器，就是因为在使用它处理交易的时候，所形成的交易数据并未真正直接写入 state 数据库中，而是将交易查、增、删得到的数据暂时放入了一个叫作读写集的地方备用，因此是模拟交易。

一个 chaincode 的每笔交易都对应一个读写集，读写集实现为 rwsetutil/rwset_builder.go 中的 RWSetBuilder，成员只有一个 rwMap map[string]*nsRWs 映射，即以 chaincodeID 为 key，每个 chaincode 单独有一个 nsRWs。读写集存储三类数据：KVRead 读值、KVWrite 写值、RangeQueryInfo 范围读值（三者均定义在 protos/ledger/rwset/kvrwset/ 下），多个值各自形成 readMap 读集、writeMap 写集、rangeQueriesMap 和 rangeQueriesKeys 组成的范围读集。分别描述如下：

1）读值：KVRead，每个读值只包含 key 和值的版本号，而不包含值本身。TxSimulator 每次调用 GetState() 接口，都会将读取到的读值写入交易的读集。

2）写值：KVWrite，每个写值包含 key 和 value，还包含一个此写值是否为删除的标识 IsDelete。TxSimulator 每次调用 SetState()、DeleteState() 等接口，都会将一个写值写入交易的写集。

3）范围读值：RangeQueryInfo，每个范围读值是一个范围内所有读值的集合，范围读值何时被写入将在下文提及。rangeQueriesMap 以 rangeQueryKey 为 map 的 key，RangeQueryInfo 为 map 的 value。rangeQueryKey 中，startKey 标识了范围从何处开始，endKey 标识了范围至何处结束（注意，范围读值中不包含 endkey 对应的值），itrExhausted 标识了是否结束。RangeQueryInfo 中，其余的成员同 rangeQueryKey 一样，而成员 ReadsInfo 存储两类数据：原始范围内的多个读值或范围内的读值的哈希值。范围内的多个原始的读值很容易理解，而范围内读值的哈希值的生成则需要借助其他工具。

当用户发起读取一定范围数据的交易时，不是直接将这个范围内所有读值集合打包返回给调用者，而是将一个包含了读值范围信息的迭代器返回给调用者（这样可以避免数据量过大而导致的处理效率低下），然后调用者使用迭代器的 Next() 一个一个抽取读值。这里涉及两个迭代器：

1）在交易模拟器的范围内，为 txmgr/lockbasedtxmgr/helper.go 中的 resultsItr，由交易模拟器的 GetStateRangeScanIterator(...) 接口获取，每获取一个迭代器，都会把迭代器记录在 queryHelper 中的 itrs。再次强调，TxSimulator 包含 QueryExecutor，而提供迭代器也是 QueryExecutor 对 TxSimulator 在查询功能上主要的拓展之一。

2）在核心的 chaincode 处理范围内，为 core/chaincode/shim/chaincode.go 中的 StateQuery-

仅供非商业用途或交流学习使用

Iterator。该迭代器的使用依附于 1）中迭代器所查询出来的结果。

（1）迭代器

resultsItr 迭代器管理了一个工具，为 rwsetutil/query_results_helper.go 中的 RangeQueryResultsHelper（亦是“人如其名”）。RangeQueryResultsHelper 中的 pendingResults 用于暂时存储范围内的读值，merkleTree 用于存储一棵默克尔树，该树存储 pendingResults 对应的哈希值，两者是随着数据的增加同步更新的（其实哈希值是否同步生成是由 HashingEnabled 决定的，而该值又由账本的配置决定，该配置默认开启，由 core/ledger/ledgerconfig/ledger_config.go 中的 IsQueryReadsHashingEnabled() 接口决定）。每次 resultsItr 迭代器被调用一次 Next()，读取的读值都会添加到 RangeQueryResultsHelper 中的 pendingResults 和 merkleTree 中暂存。

这里粗略地讲一下默克尔树，也叫哈希树。实现为 query_results_helper.go 中的 merkleTree，由工具 RangeQueryResultsHelper 持有并管理。在 merkleTree 中，一个 map[MerkleTreeLevel][]Hash 格式的映射中实现了该树，树的每个节点保存的是哈希值。默克尔树中有两个比较重要的值：Level 和 maxDegree。Level 标识树的层级，默认为 1，当树中的 Level1 有 maxDegree+1 个节点时，则会进行一个归并哈希值的操作：如两个节点存有 Hash1、Hash2，该操作是将 Hash1 和 Hash2 的值前后连成一个整体，然后对这个整体进行哈希运算，得到一个新的哈希值 Hash1-2，然后将 Hash1-2 放入上一层（也就是 Level2）里面。

（2）验证器

state 数据库是保存的最实时最新的状态，peer 节点进行查询操作时，也是从该账本读取的数据。所以在向 state 数据库提交交易数据前要使用验证器进行验证，无效的交易将被屏蔽掉而不会提交到 state 账本中。这个验证器在 validator/statebasedval/state_based_validator.go 中实现为 Validator，其 ValidateAndPrepareBatch(block,domvcc) 接口负责实现验证功能，其中第 2 个参数 domvcc 决定了是否进行 mvcc 验证，该值为 true。关于 mvcc（多版本并发控制），可自行搜索参看与数据库相关的 mvcc 的作用。

存在验证就存在对比，因为验证只能通过对比。对比的内容是 key 对应的版本号。对比的双方：甲方是 block 中携带的读集和范围读集，乙方是验证时现从 state 数据库里查询出的甲方对应的读值和范围读集 + 由甲方自带的有效交易的写集形成的升级数据包 updates。范围读集中虽说可能是哈希值，但是这些哈希值也是在原始读值（包含 key 和版本）的基础上形成的哈希值，在验证的时候，再从 state 数据库中读取同样范围内的读值集合，按照同样的方法生成一个新的哈希值。如果两个哈希值不同，则间接说明了当前 state 数据库中这个范围内的值已经有所变动，即比较哈希值其实也是在比较 key 对应的版本号。如现在验证的 block 中一个 Envelope 中的读集中有两个读值 <read key="K1", version="1">,<read key="K2", version="1">，有一个范围读值 <queryinfo,startkey=10,endkey=20,true,Hash10-20>（注意乙方的 updates 在这个例子里面没有体现，更好的例子请参看官方文档 Example sim-

仅供非商业用途或交流学习使用

ulation and validation 章节）。验证的方法只有一个：验证时依次将 K1、K2 的 version 与现从 state 数据库中现读出的 K1、K2 的版本号相比较，然后将 Hash10-20 与现从 state 数据库中读出 startkey-endkey 间的读值并按照同样的归并哈希的方法生成的新哈希值 Hash10-20' 进行比较。概括点说，就是拿之前模拟交易产生的读集中的版本号与 state 先有的相应的版本号进行比较。验证的标准只有一个：若以上所有的比较均相同，则判定此交易有效；若任何一个比较不同，则判定交易无效。验证的目的只有一个：将有效交易的写集数据放入批量升级包中，准备更新 state 数据库。

关于理解为何如此判定交易的有效性，有几点说明：

1）state 是存储当前账本所有最新有效交易的世界状态的数据库。这里的关键词是最新、有效、交易。

2）所有 block 均为串行化后的数据。state 数据库中值的版本号由 blockID+TxID 组成，blockID 为序列递增，TxID 则为交易在 block 中一批交易中的相对位置（即交易在 block.Data.Data 这个数组中的下标值，亦是相对序列递增）。因此当一个新的有效交易的数据被提交至 state 数据库中进行更新时，该值的版本号与原值的版本号必然不一样，即更新 state 数据库的值，值的版本号必然变化。

3）模拟交易读出的数据都会放入读集中。这里假定调用者在执行 chaincode 的模拟交易时，读取出来的数据会被使用。这个也很正常，既然读出来了，就有被使用的可能。比如 chaincode_example02.go 中，先查出来 A、B 账户的余额数据，并在此基础上进行的数据的加减。因此如果读集的数据失效，在此基础上得出的写集中的数据也必然无效。

4）fabric 对交易是并发并行处理的，且自模拟交易产生读写集 -> 数据返回 peer 节点 -> 送到 orderer-> 再返回 peer 的 gossip-> 再准备提交至 state 数据库。这个过程会耗费一段时间，而这段时间内，原有交易所使用的读出数据有可能已经被其他稍早点儿的交易改变。比如 Tx_1 查到的 A 的余额为 100，B 的余额为 100，A 把 50 元转给 B，则形成的读集是 A-v1，B-v1，写集是 A-50，B-150。这个读写集在到达 Validator 验证之前，经历了一段时间。就在这一段时间内，稍早点的交易 Tx_0 已经把 A 的 20 元转给了 B，A 的余额为 80（假定版本变为 v2），B 的余额为 120（假定版本变为 v2）。如果此时依然判定 Tx_1 有效，将 Tx_1 的写集提交至 state 数据库，则将会覆盖 Tx_0 的交易，此时 A 的余额仍然为 50（实际应该是 30），B 的余额仍然为 150（实际应该是 170），这样肯定出错。

5）由上一点，有人可能会产生另一个疑问：既然从 Tx_1 在交易模拟器生成读写集数据到提交前验证之间会耗费一段时间，存在稍早的交易 Tx_0 已经把 state 数据更改而使得 Tx_1 无效的风险，那么 Tx_1 在通过验证到真正提交到 state 数据库中依然存在一段时间，这期间也可能存在被稍早的交易 Tx_N 将 state 数据更改的风险，这点如何保证呢？其实不存在 Tx_N 这种情况，因为如第 2 点所说，读写集是由 block 带入的，而 block 均为

仅供非商业用途或交流学习使用

串行化依次处理的。这样只会存在两种情况：一是 Tx_0 与 Tx_1 不在同一个 block 中，则 Tx_0 所在的 block 早于 Tx_1 所在的 block，此时一定是 Tx_0 所在的 block 处理完毕之后，才会开始处理 Tx_1 所在的 block，也因此当处理到 Tx_1 开始验证时，此时从 state 数据库中取出的版本号必定是 Tx_0 提交过后的新的版本号 v2，Tx_1（版本号为 v1）会被判定为无效交易。二是 Tx_0 与 Tx_1 在同一个 block 中，则 Tx_0 所在的位置（即下标值）小于 Tx_1 所在的位置，这种情况会先验证 Tx_0 交易，如果 Tx_0 有效，则会把 Tx_0 的写集放入一个 updates 升级包中（但此时这个升级包并未提交到 state 数据库中），接着会继续验证 Tx_1，会发现 updates 中存在与 Tx_1 读集相同的 key，也即说明在 Tx_1 之前的 Tx_0 已经在试图改变 Tx_1 所使用到的数据，而 Tx_0 又是有效且早于 Tx_1 的，因此要判定 Tx_1 为无效。

（3）验证过程

学习验证的过程不仅可以了解数据的包装结构，还能很清楚地知道什么情况下是对的，什么情况下是错的，都需要哪些对象参与，等等。这些对于整个业务逻辑和流程的理解都很有帮助。其实 committer 将 block 提交至账本处时已经使用自身的验证器执行了一轮验证（参看 core/committer/txvalidator/validator.go 中的 Validate(block)），并把非法的无效交易进行记录，不过该轮验证主要集中在验证 block 的身份、签名、权限等内容。而这里所讲的是账本中对数据的 mvcc 等验证，以确保有效交易能正确存储到 state 数据库中，所以验证过程以 validator/statebasedval/state_based_validator.go 中的 ValidateAndPrepareBatch(block,domvcc) 为起点（domvcc 默认为 true）。从函数名即可理解这个函数分为上下两部分：Validate 和 PrepareBatch，即验证和准备批量升级包 updates。

1）updates := statedb.NewUpdateBatch()，准备 updates 用于存放批量升级包，这个 updates 就是上文提到的升级包，会以参数的形式传递到每个更具体的验证函数中，作为乙方的一部分参与验证 ->txsFilter := util.TxValidationFlags(...)，从 block 中抽取出 block 元数据的 BlockMetadataIndex_TRANSACTIONS_FILTER 位的数据 txsFilter，该数据标记了 block 中每笔交易的有效性（这里的有效性数据即是 committer 处验证的结果），若是无效交易，这里的验证将直接跳过。交易的有效值在 protos/peer/transaction.pb.go 中定义，如 TxValidationCode_VALID 系列常量值。

2）for txIndex, envBytes := range block.Data.Data { ... }，依次抽取 block 中的每个 Envelope，开始验证 -> 一系列解压抽取 Envelope 的工作后，if txType != common.HeaderType_ENDORSER_TRANSACTION，判定类型，若不是正常的交易数据，则直接跳过，state 只存储交易数据 -> 最重要的一步，txRWSet, txResult, err := v.validateEndorserTX(env...)，验证交易，返回交易的读写集和验证结果 ->txsFilter.SetFlag(txIndex, txResult)，在 txsFilter 中存储当前交易的验证结果，if txRWSet != nil { ... }，如果读写集不为空，则说明当前交易有效，则 committingTxHeight := version.NewHeight(...)，生成提交版本号（写值自身是不带版

本号的，且写值向 state 数据提交的版本号就是在这生成的），addWriteSetToBatch(...) 将有效交易的写集和对应的提交版本号放入批量升级包 updates->for 循环结束，block.Metadata.Metadata[..._TRANSACTIONS_FILTER] = txsFilter 替换 block 元数据的交易有效性值。

3）上一步中的 txRWSet, txResult, err := v.validateEndorserTX(env)，就是一个分发任务依次验证的过程。自身 txRWSet.FromProtoBytes(respPayload.Results) 把需要验证的读写集抽取出来后，就把任务交给了 v.validateTx(txRWSet, updates)，validateTx 分拆，又把验证读集的任务交给 validateReadSet，再把验证范围读集的任务交给 validateRangeQueries。这些函数均是“人如其名”的存在。

（4）验证读集

validateReadSet 验证读集的方法就是循环地调用 validateKVRead，一一验证每一个读值。在 validateKVRead 中：

1）if updates.Exists(ns, kvRead.Key) 即是查看升级包 updates 中是否有与读值相同的 key，若存在，则直接判定交易无效。

2）versionedValue, err := v.db.GetState(ns, kvRead.Key)->committedVersion = versionedValue.Version，现从 state 数据库中查出读值 key 在 state 中的版本号 committedVersion-> if !version.AreSame(...)，比较两个版本号是否一致，若不一致，则判定当前交易无效，直接返回 false，否则返回 true。

（5）验证范围读集

validateRangeQueries 验证范围读集的方法就是循环调用 validateRangeQuery，一一验证每个范围读值。由于范围读值的验证是一系列的值，这一系列的值既要与 state 中现存的版本号比较，也要与 updates 中已存在的 key 比较，还涉及可能范围读值中的一个 key 在 updates 和 state 中同时存在（这时这个 key 就存在两个版本号了）而选哪一个版本号来和范围读值中这个 key 的版本比较的问题，所以就稍显麻烦。为了描述简捷，下文值提及的 key，既代表 key 本身，也代表 key 对应的读值。

这里涉及 validator/statebasedval/ 下的两个工具：

1）range_query_validator.go 中的 rangeQueryValidator，范围查询验证器，有 init 接口用于初始化，validate 接口用于验证。实现有两个版本，一个是用来验证原始范围读值的 rangeQueryResultsValidator，一个是用来验证范围读值的归并哈希值的 rangeQueryHashValidator。由于多数情况下范围读值中存储的是读值集合的归并哈希值，因此这里只讲后者。

2）combined_iterator.go 中的 combinedIterator，联合迭代器。这个联合迭代器是供 rangeQueryValidator 管理使用的。针对上述验证的麻烦之处，联合迭代器联合的就是以 updates 为数据源生成的迭代器 A（参看 statedb/statedb.go 中的 nsIterator）和以 state 数据库为数据源生成的迭代器 B（参看 statedb/stateleveldb/stateleveldb.go 中的 kvScanner）。当前从 A 中获取的值存储在 updatesItem 中，当前从 B 中获取的值存储在 dbItem（类型为 VersionedKV，包含 key、值、版本号）中。当执行联合迭代器的 Next() 获取一个查询的

版本号时，联合迭代器会调用 compareKeys() 比较 updatesItem 和 dbItem 中的 key：若 updatesItem 的 key 更小，返回 updatesItem，A 前进一步；若 dbItem 的 key 更小，返回 dbItem，B 前进一步；当两个 key 相等时返回 updatesItem，A、B 都前进一步。这里的前进一步指的是迭代器执行一下 Next() 接口，迭代到下一个值。

回到 validateRangeQuery 中。

1）includeEndKey := !rangeQueryInfo.ItrExhausted，记录下此范围读值是否包含 endkey 对应的读值。因为范围读值默认是不包含 endkey 对应的读值的，但是当这个范围读值没有被模拟器读尽时，比如 key2 ～ key10 只读了前 3 个就返回了，此时 rangeQueryInfo.ItrExhausted 值为 false，说明此时实际的 endkey 是 key4 而非 key10，且此时范围读值应该包含 key4 的读值。相反，若 ItrExhausted 值为 true，则说明模拟器读尽了 key2 ～ key10 间的 8 个 key，由于范围读值默认的不包含 endkey，所以此时范围读值中不包含 key10 对应的读值。

2）combinedItr, err := newCombinedIterator(...)，根据 updates、startkey、endkey、includeEndKey 创建一个联合迭代器。

3）if rangeQueryInfo.GetReadsMerkleHashes() != nil 成立（默认返回的范围读值中是归并后的哈希值）->validator = &rangeQueryHashValidator{}->validator.init(rangeQueryInfo, combinedItr)，根据范围读值 rangeQueryInfo 和生成的联合迭代器 combinedItr，同时内建了一个上文提及的 RangeQueryResultsHelper 工具，创建一个用于验证范围读值的归并哈希值的验证器。

4）validator.validate()，使用验证器验证范围读值。粗略的验证过程就是使用 RangeQueryResultsHelper 工具重新一步步构建出一个与 rangeQueryInfo 范围相同的默克尔树，并不断进行归并哈希的操作，将得到的归并哈希值与 rangeQueryInfo 携带的哈希值进行比较。

回到 state_based_validator.go 中的 ValidateAndPrepareBatch(...)，当 validateEndorserTX 调用的 v.validateTx(txRWSet, updates) 返回为 peer.TxValidationCode_VALID，则说明经过上述验证读集、验证范围读集的过程，证明当前交易有效且可以写入 state 数据库。因此会把交易的读写集以不为 nil 的形式返回，至此 ValidateAndPrepareBatch 的 Validate 部分的工作算是完成。接着，在 ValidateAndPrepareBatch(...) 中会进入 if txRWSet != nil {...} 分支，完成 PrepareBatch 部分的工作。最后，将准备的升级包 updates 返回。该 updates 返回后会被存储给 LockBasedTxMgr 的 batch 成员，供下一步 l.txtmgmt.Commit() 向 state 数据提交这些升级数据的时候使用。在 l.txtmgmt.Commit() 中（txmgr/lockbasedtxmgr/lockbased_txmgr.go），txmgr.db.ApplyUpdates(...) 即是 state 数据库使用升级包数据来真正升级状态的。这里传入了两个参数，一个是 txmgr.batch，即升级数据包，另一个是以 blockID+block 最后一个交易的下标序号组成的版本号 s_version。

（6）state 数据库

state 数据库就是一个正常的 leveldb 数据库，实现为 statedb/stateleveldb/stateleveldb.go

中的 versionedDB。提供基本的打开、关闭、查询、写入功能，此外还提供支持范围查询的 leveldb 数据库的迭代器、多值查询等额外功能。leveldb 版本的 state 数据不支持富查询，即 ExecuteQuery(...) 接口直接返回错误。

数据库查操作如下：

GetState、GetStateMultipleKeys 两个接口分别提供单值查询、多值查询。GetStateRangeScanIterator 接口提供范围查询的迭代器。这些较为简单，不赘述。

数据库写操作如下：

state 数据库为 LockBasedTxMgr 处理 block，因此只支持批量升级数据，即 ApplyUpdates (batch,height) 接口的功能。batch 即为上文验证过程中准备的有效交易的升级数据包，height 则为一个版本号（即上文的 s_version），该版本号不用于具体的某个状态，而是作为 state 数据库的一个叫作保存点的键值对的值（idStore、BlockStore 中都有类似的保存点，也可以叫作检查点）。具体的写入操作也是常规的 leveldb 数据库的批量写入操作，保存点会在每一批升级数据的最后加入，key 为 savePointKey，value 为 height。也就是说，当能从 state 中获取的保存点的 height 值，且 height 中的 BlockNum 为 N 时，则说明当前 state 数据库已经完整保存了序号为 N 的 block 中的有效交易。

数据库重启恢复操作如下：

同 idStore 和 BlockStore 存储一样，VersionedDB 同样存在宕机重启后残缺数据的清理和恢复问题。idStore 和 BlockStore 在自身对象建立起来后即进行“自我修复”，而 VersionedDB 需要在 BlockStore 自我修复完毕之后，使用恢复后的 BlockStore 提供的数据，该数据由 kvLedger 手动恢复。为何使用 BlockStore 提供的数据来恢复呢？这点需要参看账本处理 block 的前后顺序：参看 core/ledger/kvledger/kv_ledger.go 的 Commit(block)，是先执行 l.blockStore.AddBlock(block) 把 block 提交至 BlockStore 中后，再执行 l.txtmgmt.Commit() 并将有效交易数据提交至 state 数据库，最后向 HistoryDB 账本提交（这个提交顺序很重要，对恢复各个账本时的逻辑有很大影响）。BlockStore 亦用于自恢复能力，在 newKVLedger(...) 时，传入的 BlockStore 对象其实已经完成了自我修复。因此当宕机发生时，BlockStore 中的数据可能比 VersionedDB 的内容更新。在 newKVLedger(...) 中，随后执行了 l.recoverDBs()，以恢复 VersionedDB 和 HistoryDB，HistoryDB 在后面详述。

在 recoverDBs() 中，lastAvailableBlockNum 是从 BlockStore 中获取的最新的有效的 blockID，recoverables 存放预计需要恢复的账本对象，recoverers 存放真正需要恢复的账本对象。

1）info, _ := l.blockStore.GetBlockchainInfo()，lastAvailableBlockNum := info.Height-1，使用已恢复的 BlockStore 获取当前已经写入 BlockStore 的有效的 blockID。

2）recoverables := []recoverable{...}，可能需要恢复 VersionedDB 和 HistoryDB 两个数据库。

3）for _, recoverable := range recoverables {...}，依次调用 recoverable.ShouldRecover

(lastAvailableBlockNum)，使用最新有效的 lastAvailableBlockNum，检测账本是否需要恢复。这里只看 VersionedDB（HistoryDB 的检测方式与之相同）。在 txmgr/lockbasedtxmgr/lockbased_txmgr.go 的 ShouldRecover 中，savepoint, err := txmgr.GetLastSavepoint() 获取了 state 数据库中的保存点 -> 将 savepoint.BlockNum 与 lastAvailableBlockNum 进行对比 -> 如果相等，则 state 数据库不需要恢复，如果不同（savepoint.BlockNum 一定比 lastAvailableBlockNum 小），则说明 state 中未完整存储序号为 lastAvailableBlockNum 的 block 的所有有效交易，需要进行恢复 -> 当需要恢复时，将返回 ture，savepoint.BlockNum+1。true 表示 state 需要恢复操作，savepoint.BlockNum+1 则标识需要从哪一块 block 进行恢复，即恢复起点。ShouldRecover 结束后，返回的数据支付给了 recoverFlag，firstBlockNum，如果 recoverFlag 为 true，则会将 VersionedDB 账本放入 recoverers。这里假设 VersionedDB 确实需要恢复。

4）for 循环之后，有一系列的 if 分支。if len(recoverers) == 0，说明没有确实需要恢复的账本，直接返回；if len(recoverers) == 1，则说明只有一个需要恢复的账本，此时只可能是 HistoryDB，因为若 VersionedDB 需要恢复，则 HistoryDB 必然也需要恢复，且 VersionedDB 的恢复起点一定 >= HistoryDB 的恢复起点。因此这里只看两个账本都需要恢复的情况，设 VersionedDB 需要从 10 处开始恢复，HistoryDB 需要从 6 处开始恢复，lastAvailableBlockNum 值为 15，则：① if [0]...>[1]... 将成立，因此执行 [0], [1] = [1], [0] 置换，将 lagger 放入 recoverers 的 0 的位置，lagger 这里指恢复起点值更小的账本，且肯定是 HistoryDB，而把恢复起点值更大的 VersionedDB 放到后边。② if [0]... != [1]...，置换后，0 处的 6 肯定 != 1 处的 10，因此执行 l.recommitLost-Blocks(...)，单独向 lagger 提交 6 ～ 9 之间的 block 数据进行恢复。③ l.recommitLost-Blocks([1].., lastAvailableBlockNum,[0],[1])，最后，向 lagger 和 VersionedDB 一同提交 10 ～ 15 之间的 block 数据进行恢复。当两个账本的恢复起点相等时，比如都为 8，则①和②处的 if 分支都不会进入而直接进行至此步，一起向两个账本提交 8 ～ 15 之间的 block 数据进行恢复。这里需要说明一下，当两个账本的恢复起点不一致时为什么要进行①和②的操作，又是换位置又是分段恢复的：因为 block 是存储在磁盘文件 blockfile 中的，因此换位置和分段恢复的操作可以使得恢复过程中 BlockStore 读取 blockfile 中的 block 数据时顺序读取一次，这样效率更高。

5）最后单独看一下 recommitLostBlocks(...)，传入的参数分别为恢复起点 firstBlockNum、恢复终点 lastBlockNum、需要恢复的账本 recoverables。for 循环中，block, err = l.GetBlockByNumber(blockNumber) 使用 BlockStore 从 blockfile 中获取指定序列号的 block，然后调用账本的 r.CommitLostBlock(block)，向账本提交 block，进行恢复。这里只看 VersionedDB。在 txmgr/lockbasedtxmgr/lockbased_txmgr.go 的 CommitLostBlock(block) 中，和正常的向 VersionedDB 账本提交交易数据的过程如出一辙：txmgr.ValidateAndPrepare(block, false) 准备升级包数据，注意这里第 2 个参数给的是 false，即不再做 mvcc 验证，因为这里是恢复，所提交的 block 是从 BlockStore 中获取的，也肯定是之前已经验证过才会放入 BlockStore 的，所以这里不需要再重复进行验证 ->txmgr.Commit()，将升级包数据真正提交到 state 数

据库，完成 VersionedDB 的恢复工作。

4. HistoryDB

HistoryDB 也是一个标准的以 leveldb 数据库为依托的账本，实现在 core/ledger/kvledger/history/historydb 下（后面以此目录为基准）。比较特殊的地方是，这个账本只存储 block 中与有效交易相关的 key，而不存储 value（由于 leveldb 存储的键值对，不允许 nil，因此这里实际上 value 是有值的，只不过所有的值均为 []byte{}）。另外提供一个历史查询器 HistoryDBQueryExecutor 在交易的时候使用。其余的操作，如开闭、读写操作均与正常的 leveldb 类型账并并无二致，不赘述。

HistoryDB 是可配置“是否使能”的，在 core/ledger/ledgerconfig/ledger_config.go 的 IsHistory-DBEnabled() 接口可以获取当前是否开启使用了 HistoryDB。原始的配置则在 core.yaml 中的 enableHistoryDatabase 项，默认为 true，即开启使用 HistoryDB。

既然 HistoryDB 实际只存储 key，那么，key 自身既携带了索引信息又携带了我们需要的值信息，key 其实就是 HistoryDB 所要存储等价于 value 的对象并供外界检索。这个 key 是一个组合 key，对照 historyleveldb/historyleveldb.go 中的 Commit(block) 接口，当向 HistoryDB 提交一个 block 时，会筛选出 block 中的有效交易，并把这些交易的写集中的每个写值读取出来，以命名空间 ns+compositeKeySep+ 写值 key+compositeKeySep+block 序列号 + 交易 ID 的组合形式形成一个组合键 compositeHistoryKey，然后 dbBatch.Put-(compositeHistoryKey, emptyValue)，将 compositeHistoryKey 与空值 emptyValue 作为一个键值对写入批量升级包 dbBatch 中。当 block 中所有有效交易均遍历完毕后，dbBatch.Put(savePointKey, height.ToBytes()) 以保存点封底，最后 historyDB.db.WriteBatch(dbBatch, false) 向 HistoryDB 提交数据。

组合键 compositeHistoryKey 中的 compositeKeySep 当作分隔符理解即可。前半部分命名空间 ns+compositeKeySep+ 写值 key+compositeKeySep 即为用于索引的信息，后半部分 block 序列号 + 交易 ID 即为值信息。

（1）历史查询器

对 HistoryDB 的检索主要通过 HistoryDB 提供的一个 HistoryDBQueryExecutor 对象来实现。HistoryQueryExecutor 在 historyleveldb/historyleveldb_query_executer.go 中实现，只提供 GetHistoryForKey(...) 一个接口，该接口根据提供的命名空间和写值 key，返回一个迭代器 historyScanner（内部封装了 levedb 数据库的 Iterator）。迭代器必定就涉及起点和止点，historyScanner 的起点是命名空间 ns+compositeKeySep+ 写值 key+compositeKeySep 的组合键，止点是命名空间 ns+compositeKeySep+ 写值 key+compositeKeySep+0xff 的组合键。对比起点和止点，止点多了一个 0xff，相当于一个字符的极限值，也因此这个范围查询的是所有以命名空间 ns+compositeKeySep+ 写值 key+compositeKeySep 为开头的 key 值。比如起点是 []byte("A")，止点是 append([]byte("A"),[]byte{0xff}...)，则这个范围查询的是所有以字符 A 开头的 key。又因为有 HistoryDB 存储的格式为前提，假设这里的 ns 为“ chaincode_

example02”，写值 key 为“A 账户”，则这个起止范围相当于在查询所有 chaincode_example02 链码上改动过 A 账户的“blockID+ 有效交易 ID”的信息。而有了 blockID+ 有效交易 ID 这两个信息，自然可以通过 BlockStore 定位查询出原交易的所有信息，如改动时间、改动值、是否是删除操作等。

参看 historyleveldb_query_executer.go 中 historyScanner 的 Next() 接口：

1）if !scanner.dbItr.Next()，因为 HistoryDB 不存储 value，因此这里 leveldb 的迭代器 dbItr 的 Next() 操作不为获取 value，而只是让迭代器前进一步，同时进行是否还有下一个值的判断。

2）historyKey := scanner.dbItr.Key()，获取 leveldb 迭代器当前值的 key，这也是我们实际想获取的内容，这个 key 就是由命名空间 ns+compositeKeySep+ 写值 key+compositeKeySep+block 序列号 + 交易 ID 组成的组合键。

3）blockNum, bytesConsumed := util...，tranNum, _ := util...，从组合键中分别分解出其中携带的 blockID 和交易 ID。

4）tranEnvelope, err := scanner.blockStore...，使用 BlockStore，根据 blockID 和交易 ID，获取原始的交易信息。

5）queryResult, err := getKeyModificationFromTran(...)，根据获取的原始交易信息进行整理，返回当次 Next() 的单个查询结果。该结果里面包含改动时间、改动值、是否是删除操作等信息。

（2）使用

HistoryDB 账本被 kvLedger 使用的过程和 VersionedDB 在过程上就是一先一后的区别。被 chaincode 使用主要是通过出现在交易中的 HistoryDBQueryExecutor 进行查询，而查询的过程与 VersionedDB 的交易模拟器的范围查询过程颇为类似（至于 HistoryDBQueryExecutor 为何会出现在交易中，请参看 core/endorser/endorser.go 的 ProcessProposal(...) 中的 ctx = context.WithValue(...) 处）。以 map.go 为例。

1）peer chaincode invoke...，在 peer 节点执行 map 的 Invoke 命令，中间过程省略，直接定位到下一步。

2）在 map.go 的 Invoke(stub) 的 case "history": 分支中，keysIter, err := stub.GetHistoryForKey(key)，调用 ChaincodeStubInterface 的此接口，最终获取一个迭代器，该迭代器以 HistoryDBQueryExecutor 获取的范围数据为数据源，返回查询结果。

3）第 1）步调用后，会在 ShimHandler 和 ServerHandler 间辗转，然后在 core/chaincode/handler.go 中调用 handleGetHistoryForKey(...)，在这个函数中，historyIter, err := txContext.historyQueryExecutor.GetHistoryForKey(chaincodeID, getHistoryForKey.Key) 即是使用 HistoryDBQueryExecutor 获取一个 historyScanner，然后在 getQueryResponse(...) 中依次调用 historyScanner 的 Next() 获取查询结果（即上面第 5 步查询到的内容）。查询结果最终放入 ChaincodeMessage 的 Payload，返回。

4）携带查询结果的 ChaincodeMessage 返回至 core/chaincode/shim/chaincode.go 中的 GetHistoryForKey(...) 中，即第 1）步所调用处，然后把结果集放在了 CommonIterator 迭代器中，上面再套上 HistoryQueryIterator 返回给 chaincode，也就是第 1 步获取的 keysIter。

5）获取 keysIter 后，就可以在 for keysIter.HasNext() { response, iterErr := keysIter.Next()... } 进行使用，以完成 chaincode 在该功能上自身想完成的任务。

因此，从使用上讲，HistoryDB 算是一个辅助性的数据库，辅助其他数据库，辅助交易的进行。另外，拓展一下的话，HistoryDB 的组合键的组合形式可以针对自身业务的需求进行设计，这样也可实现类似 CouchDB 数据库那样的富查询。

5.6　加密服务

加密涉及的内容挺复杂，是一门专业性很强的学科。在此只是列举一些 bccsp 服务所涉及的概念。

- RSA，一种非对称的加密算法，用于加密。有几种族簇，如 RSA1024、RSA2048 等。
- AES，一种块加密算法，用于加密成块的大量数据。有几种族簇，如 AES128、AES192 等。
- ECDSA，一种椭圆曲线签名，用于签名。有几种族簇，如 ECDSAP256、ECDSAP384 等。
- Hash，哈希算法，有几种族簇，如 SHA_256、SHA3_256 等。
- HMAC，密匙相关的哈希运算消息认证码。
- x509，证书的一种，是国际电联电信委员会为单点登录和授权管理基础设施而制定的。
- PKCS#11，一套标准安全接口，可与安全硬件相关，以上的这些东西，可以用它来建立、读取、写入、修改、删除等操作进行管理。

Fabric 所用到的这些技术的常量名称在 /fabric/bccsp/opts.go 中开始的部分定义，如 ECDSA 支持 ECDSAP256、ECDSAP384 等几种类型。

Fabric 加密服务由 BCCSP 提供，BCCSP 是 BlockChain Cryptographic Service Provider 的缩写，可译作区块链加密服务提供者，为 Fabric 项目提供各种加密技术、签名技术。BCCSP 的工具性很强，MSP 服务模块中就使用到了 BCCSP。这里需要说明的一点是，工具性强说明了如果不是想专门学这一领域，就不用太在乎其实现的细节，只要用就行了。BCCSP 服务的代码集中在 /fabric/bccsp 中，目录结构如下：

1）mocks- 模拟代码文件夹，可以参看之帮助理解 bccsp 服务；

2）signer- 实现的是 crypto 标准库的 Signer 接口；

3）factory-bccsp 服务工厂；

4）pkcs11-bccsp 服务实现之一，HSM 基础的 bccsp（the hsm-based BCCSP implementation）；

5）sw-bccsp 服务实现之二，软件基础的 bccsp（the software-based implementation of the BCCSP）；

6）utils-bccsp 服务工具函数；

7）bccsp.go- 定义了 BCCSP、Key 接口，众多 BCCSP 接口所使用到的选项接口，如 Key、KeyGenOpts、KeyDerivOpts 等；

8）keystore.go- 定义了 key 的管理存储接口，如果生成的 key 不是暂时的，则存储在该接口的实现对象中；如果是暂时的，则不存储；

9）XXXopts.go-XXX 表示该目录下的各种值，bccsp 服务实现所使用到的各种技术的选项实现。

从以上可以看出，bccsp 服务有两种实现：pkcs11 和 sw。pckcs11 是硬件基础的加密服务实现，sw 是软件基础的加密服务实现。这个硬件基础的实现以 https://github.com/miekg/pkcs11 这个库为基础，而 HSM 是 Hardware Security Modules，即硬件安全模块的缩写。相对应的两种 bccsp 服务实现，这里有两种工厂，两种工厂为其他使用 bccsp 服务的模块提供了窗口函数（就是给其他模块提供窗口的函数，这些函数一般统一管理自己服务的功能模块，供外界调用，如后面所讲的 InitFactories 函数）。所有产生的 bccsp 实例存储在 factory/factory.go 中所定义的全局变量中。

5.6.1 BCCSP 的接口和选项

（1）接口

```
//在fabric/bccsp/bccsp.go中定义
type BCCSP interface {
    //根据key生成选项opts生成一个key
    //与key有关的选项opts选项要适合原始的key（与"证书是一级一级的认证"对看）
    KeyGen(opts KeyGenOpts) (k Key, err error)
    //根据key获取选项opts从k中重新获取一个key
    KeyDeriv(k Key, opts KeyDerivOpts) (dk Key, err error)
    //根据key导入选项opts从一个key原始的数据中导入一个key
    KeyImport(raw interface{}, opts KeyImportOpts) (k Key, err error)
    //根据SKI返回与该接口实例有联系的key
    GetKey(ski []byte) (k Key, err error)
    //根据哈希选项opts哈希一个消息msg，如果opts为空，则使用默认选项
    Hash(msg []byte, opts HashOpts) (hash []byte, err error)
    //根据哈希选项opts获取hash.Hash实例，如果opts为空，则使用默认选项
    GetHash(opts HashOpts) (h hash.Hash, err error)
    //根据签名者选项opts，使用k对digest进行签名，注意如果需要对一个特别大的消息的hash值
    //进行签名，调用者则负责对该特别大的消息进行hash后将其作为digest传入
    Sign(k Key, digest []byte, opts SignerOpts) (signature []byte, err error)
    //根据鉴定者选项opts，通过对比k和digest，鉴定签名
    Verify(k Key, signature, digest []byte, opts SignerOpts) (valid bool, err error)
    //根据加密者选项opts，使用k加密plaintext
    Encrypt(k Key, plaintext []byte, opts EncrypterOpts) (ciphertext []byte, err error)
```

```
    //根据解密者选项opts，使用k对ciphertext进行解密
    Decrypt(k Key, ciphertext []byte, opts DecrypterOpts) (plaintext []byte, err error)
}
```

（2）选项

bccsp 文件夹中任何带 opt 字眼的文件，都是与选项有关的源码。关于对象的配套选项，在介绍 MSP 服务时就讲过。根据一个选项的不同配置，对象主体可以得到不同的数据或进行不同的操作，这也是一种比较值得学习的语言上的组织技巧，尤其是 bccsp 这种涉及技术比较多，而每个对象自身又分为好多类的情况。在此以哈希选项 HashOpts 和 key 导入选项 KeyImportOpts 作为例子进行说明。

```
// 在 /fabric/bccsp/bccsp.go 中定义
// 哈希选项接口
type HashOpts interface {
    Algorithm() string   // 获取 Hash 算法字符串标识，如 "SHA256", "SHA3_256"
}
// 在 /fabric/bccsp/Hashopts.go 中定义
// 哈希选项实现之一，SHA256 选项
type SHA256Opts struct {
}
func (opts *SHA256Opts) Algorithm() string {
    return SHA256
}
// 哈希选项实现之二，SHA384 选项
type SHA384Opts struct {
}
func (opts *SHA384Opts) Algorithm() string {
    return SHA384
}
-------------------------------------------------
// 在 /fabric/bccsp/bccsp.go 中定义
// key 导入选项接口
type KeyImportOpts interface {
    Algorithm() string   // 返回 key 导入算法字符串标识
    Ephemeral() bool     // 如果生成的 key 是短暂的 (ephemeral)，返回 true，否则返回 false
}
// 在 /fabric/bccsp/opts.go 中定义
// key 导入选项接口实现之一，ECDSA 公匙的导入选项
type ECDSAPKIXPublicKeyImportOpts struct {
    Temporary bool
}
func (opts *ECDSAPKIXPublicKeyImportOpts) Algorithm() string {
    return ECDSA
}
func (opts *ECDSAPKIXPublicKeyImportOpts) Ephemeral() bool {
    return opts.Temporary
}
// key 导入选项接口实现之二，ECDSA 私匙的导入选项
```

```
type ECDSAPrivateKeyImportOpts struct {
    Temporary bool
}
func (opts *ECDSAPrivateKeyImportOpts) Algorithm() string {
    return ECDSA
}
func (opts *ECDSAPrivateKeyImportOpts) Ephemeral() bool {
    return opts.Temporary
}
// 比较特殊的，比如签名选项接口 SignerOpts
// 因为使用的是标准库，因此使用到此选项时多赋值为 nil，bccsp 源码中未实现
```

5.6.2 SW 实现方式

BCCSP 的 SoftWare（SW）实现形式是默认的形式，这点仅从 /fabric/bccsp/factory/opts.go 中工厂的默认选项 DefaultOpts 和核心配置文档中关于 bccsp 的配置就可以看出来。主要使用的包是标准库 Hash 和 crypto（包括其中的各种包，如 aes、rsa、ecdsa、sha256、elliptic、x509 等）。

目录结构如下：

/fabric/bccsp/sw

impl.go-bccsp 的 sw 实现 impl。
internals.go- 签名者、鉴定者、加密者、解密者接口定义。
conf.go-bccsp 的 sw 实现的配置定义。

---

aes.go-aes 类型的生成 key 函数、加密者 / 解密者实现。
ecdsa.go-ecdsa 类型的签名者、公匙 / 私匙鉴定者实现。
rsa.go-rsa 类型的签名者、公匙 / 私匙鉴定者实现。

---

aeskey.go-aes 类型的 Key 接口实现。
ecdsakey.go-ecdsa 类型的 Key 接口实现。
rsakey.go-rsa 类型的 Key 接口实现。

---

dummyks.go-dummy 类型的 KeyStore 接口实现 dummyKeyStore，若生成的 key 是短暂的，则说明这些 key 不会保存到文件中，而是保存到内存中，系统一关闭，这些 key 就消失了。
fileks.go-file 类型的 KeyStore 接口实现 fileBasedKeyStore，若生成的 key 不是短暂的，则说明这些 key 在导入时会存储在文件中，即便系统关闭，这些 key 也不会消失。

BCCSP 接口实现如下：

```
//在/fabric/bccsp/sw/impl.go中定义
//SW bccsp的实例结构体
type impl struct {
    conf *config                          //bccsp实例的配置
    ks   bccsp.KeyStore                   //key存储系统对象，存储和获取Key对象
    encryptors map[reflect.Type]Encryptor //加密者映射
    decryptors map[reflect.Type]Decryptor //解密者映射
    signers    map[reflect.Type]Signer    //签名者映射，Key实现的类型作为映射的键
    verifiers  map[reflect.Type]Verifier  //鉴定者映射，Key实现的类型作为映射的键
}
//专用生成函数
func New(...) (bccsp.BCCSP, error) { ... }
//接口实现
func (csp *impl) KeyGen(opts bccsp.KeyGenOpts) (k bccsp.Key, ...) { ... }
...
```

粗线条上看，由于 impl 对象和各种操作选项的存在，绝大部分接口的实现都以 switch-case 为主干，根据选项的类型或配置，分情况完成功能。且每个分支的操作基本都很类似，如 KeyGen。接下来一一介绍。

（1）KeyGen

根据 key 生成选项不同，生成 3 种系列的 key：ECDSA，AES，RSA。ECDSA 使用库 ecdsa 的 GenerateKey 函数，AES 使用自定义的 GetRandomBytes 函数（在 aes.go 中实现，包装了 rand.Read），RSA 使用库 rsa 的 GenerateKey 函数。每个系列又根据具体参数的不同生成不同“尺寸”的 key。最后返回不同的 Key 实现对象，如 ecdsaPrivateKey、aesPrivateKey 等（分别定义在同目录中的 XXXkey.go 中）。这里要注意的是，当前版本中用于签名的 key，只支持 ECDSA 系列的 key，这是官方文档中所说的。但是从实现上看签名的支持不止一种，而且，bccsp 本质上无论实现多少种 key，也只能由用者决定使用哪一种。MSP 模块是 bccsp 的使用者之一，这里只认 ECDSA 的 key。

（2）KeyDeriv

根据 key 获取的选项 opts 从 k 中重新获取一个 key，这里的重新获取可以理解为把 k 中的内容重新打乱再生成一个 key。生成两种 key 类型：ecdsaXXXKey（XXX 代表 Public 和 Private）和 aesPrivateKey。最后返回打乱后重新生成的两种类型的 key：reRandomizedKey 和 hmacedKey。对于重新生成的 key，如果选项中指定的不是暂时性的 key，则会在 ks 中存储。

（3）KeyImport

从原始的数据 raw 中取出选项 opts 指定的 key，如果不是临时性的，则在 ks 中存储，最后返回 key。这里的原始指的是 byte 或者含有 key 的数据（如证书数据里的 key）。raw 是一个空接口，也就是说可以接收任何形式的原始数据。为了得到 key，对原始数据 raw 的转化或抽取，一部分使用了 utils 下的工具函数 utils.Clone 和 utils.DERToPublicKey，如 AES256、ECDSAPrivateKey 等类型的 key；另一部分直接用 Go 语言的断言 raw.(*Key 类

型），如 ECDSAGoPublicKey、RSAGoPublicKey 等类型的 key。

（4）Hash

根据哈希选项 opts 对一个消息 msg 进行哈希运算，返回该 msg 的哈希值。如果 opts 为空，则使用默认选项。这个比较好理解，算法种类支持 SHA2、SHA3 哈希家族。

（5）Sign

根据签名者选项，使用 k 对 digest 进行签名，这里的签名者选项在当前版本里没什么用处，调用者给的都是 nil。签名涉及 Key 接口和签名者 Signer 在 SW bccsp 中的实现。Key 接口有 3 种实现：ecdsakey.go 中的 ecdsaPublicKey/ecdsaPrivateKey，rsakey.go 中的 rsaPublicKey/rsaPrivateKey，aeskey.go 中的 aesPrivateKey。这里签名（自然）使用的都是私匙。签名者接口 Signer 定义在同目录下的 internals.go 中，有两种类型的实现：ecdsa.go 中的 ecdsaSigner 和 rsa.go 中的 rsaSigner。参看 SW bccsp 专用生成函数 New 的 signers 赋值部分，可知用到了两种对应类型的 Key 和 Signer：ecdsaPrivateKey-ecdsaSigner 和 rsaPrivateKey-rsaSigner。这里签名的实现是，从该 bccsp 实例的签名者集合成员 signers 中获取类型为 reflect.TypeOf(k) 的签名者 signer，然后直接调用 signer 的接口 Sign，追溯，ecdsaSigner 使用 ecdsa 库的 Sign 函数，rsaSigner 使用 rsa 库 PrivateKey 结构体的 Sign 函数。

（6）Verify

根据鉴定者选项 opts，通过对比 k 和 digest，鉴定签名。鉴定涉及 Key 接口和鉴定者接口 Verifier 在 SW bccsp 中的实现。Key 接口实现如上描述。鉴定者接口 Verifier 定义在同目录下的 internals.go 中，有两种类型的实现：ecdsa.go 中的 ecdsaPublicKeyKeyVerifier/ecdsaPrivateKeyVerifier 和 rsa.go 中的 rsaPublicKeyKeyVerifier/rsaPrivateKeyVerifier。参看 SW bccsp 专用生成函数 New 的 verifiers 赋值部分，可知所有鉴定者（与对应的 Key 接口实现）都用到。这里鉴定的实现是，从该 bccsp 实例的鉴定者集合成员 verifiers 获取类型为 reflect.TypeOf(k) 的鉴定者 verifier，然后直接调用 verifier 的接口 Verify，追溯，ecdsaXXXKeyVerifier 使用 ecdsa 库的 Verify 函数，rsaXXXKeyVerifier 使用 rsa 库的 VerifyPSS 函数（这里 XXX 表示 PublicKey 或 Private）。

（7）Encrypt

根据加密者选项 opts，使用 k** 加密 plaintext。加密涉及 **Key 接口和加密者接口 Encryptor 在 SW bccsp 中的实现。Key 接口实现如上描述。加密者接口 Encryptor 定义在同目录下的 internals.go 中，在 aes.go 中实现（只能是 aes，因为只有 aes 是用来加密的）：aescbcpkcs7Encryptor。参看 SW bccsp 专用生成函数 New 的 encryptors 赋值部分，只有 aesPrivateKey-aescbcpkcs7Encryptor 被使用。这里加密的实现是，从该 bccsp 实例的加密者集合成员 encryptors 获取类型为 reflect.TypeOf(k) 的加密者 encryptor，然后直接调用 encryptor 的接口 Encrypt，追溯，aescbcpkcs7Encryptor 使用了 aes 库的加密流程进行加密。

（8）Decrypt

解密与加密类似，过程类似，也是只有 aes 配套实现，最终使用 aes 库解密流程进行解密。

（9）GetXXX

GetXXX 系列接口，获取实例对象中的数据。

5.6.3　PKCS11 实现方式

BCCSP 的 pkcs11 实现形式主要使用的库与 sw 实现如出一辙，但外加一个 github.com/miekg/pkcs11 库，最好参看其文档以熟悉 pkcs11 的简要操作。pkcs11（PKCS，Public-Key Cryptography Standards）是一套非常通用的接口标准，可以说这里是用 pkcs11 实现了 bccsp 的功能，也为 Fabric 支持热插拔和个人安全硬件模块提供了服务。这点可以从 bccsp 的 pkcs11 的实现实例的专用生成函数 New 中所调用的 laodLib 函数看出来：laodLib 加载了一个系统中的动态库，能加载系统的动态库，就可以和驱动、热插拔、连接电脑的字符设备联系在一起。比如一款在区块链上类似于现在网上银行所用的 U 盾之类的个人身份或安全交易硬件模块或芯片，这些硬件模块或芯片也只需要遵循 pkcs11，Fabric 即可对此进行支持和扩展。

对于 pkcs11 的所提供的接口，在此提供以下两个文档网址，读者可以稍做了解：

https://www.ibm.com/developerworks/cn/security/s-pkcs/

http://docs.oracle.com/cd/E19253-01/819-7056/6n91eac56/index.html#chapter2-9

这些文档和 Fabric 都与区块链无关，但是因为 pkcs11 是通用的接口，所以有一定参考价值。pkcs11 库中的解释相对过于简单。

目录结构如下：

/fabric/bccsp/pkcs11

impl.go-bccsp 的 pkcs11 实现 impl。

conf.go- 定义了 bccsp 服务的配置和 PKCS11Opts、FileKeystoreOpts、DummyKeystoreOpts 选项。

pkcs11.go- 以 pkcs11 库为基础，包装各种 pkcs11 功能，实现了 impl 基于 pkcs11 的内调函数和一些 bccsp 服务用到的独立的内调函数。

---

aes.go- 实现 aes 类型的生成 key、加密、解密函数。

ecdsa.go- 实现 impl 在 ecdsa 技术下的签名函数 signECDSA 和鉴定函数 verifyECDSA。

---

aeskey.go- 实现 aes 类型的 Key 接口，只实现私匙 aesPrivateKey。

ecdsakey.go- 实现 ecdsa 类型的 Key 接口，实现了公匙 ecdsaPublicKey 和私匙 ecdsaPrivateKey。

rsakey.go- 实现了 rsa 类型的 Key 接口，实现了公匙 rsaPublicKey 和私

匙 rsaPrivateKey。

---

dummyks.go-dummy 类型的 KeyStore 接口实现 DummyKeyStore。

fileks.go-file 类型的 KeyStore 接口实现 FileBasedKeyStore。

BCCSP 接口实现：

```
type impl struct {
    conf *config                        //配置
    ks   bccsp.KeyStore                 //key存储系统对象，存储和获取Key对象
    ctx      *pkcs11.Ctx                //标准库的pkcs11上下文
    sessions chan pkcs11.SessionHandle //实质是uint，会话标识符频道，默认10个缓存
    slot     uint                       //安全硬件外设连接插槽标识号
    lib          string                //加载库所在路径
    noPrivImport bool                   //禁止导入私匙标识
    softVerify   bool                   //使用软件方式鉴定签名标识
}
//专用生成函数
func New(opts PKCS11Opts, keyStore bccsp.KeyStore) (bccsp.BCCSP, error) { ... }
//接口实现
func (csp *impl) KeyGen(opts bccsp.KeyGenOpts) (k bccsp.Key, err error) { ... }
...
```

BCCSP 的 pkcs11 实现的框架在 impl.go 中与 sw 的实现基本一致，只是追溯到最终实现的语句时，sw 实现使用 crypto 库下的各个包进行签名、加密、密匙导入等，而 pkcsll 则用 pkcs11 包对数据进行了多一层的处理，使用 pkcs11 提供的上下文（pkcs11.Ctx）和会话（SessionHandle）之上对签名、密匙、加密等进行管理，这也是 pkcs11.go 文件的作用。两者最大的不同是 SW 面向软件，pkcsll 面向硬件。pkcs11 自身非常冗杂，因此在此只讲 pkcs11 中与安全硬件模块建立连接的 loadLib 函数。

loadLib 函数在 pkcs11.go 中定义，供专用生成函数 New 使用。为建立与安全硬件模块的通信，进行了如下步骤：

1）根据所给的系统动态库路径 lib 加载动态库（如 openCryptoki 的动态库），调用 pkcs11.New(lib) 建立 pkcs11 实例 ctx。ctx 相当于 Fabric 与安全硬件模块通信的桥梁：bccsp<->ctx<-> 驱动 lib<-> 安全硬件模块，只要驱动 lib 是按照 pkcs11 标准开发即可。

2）ctx.Initialize() 进行初始化。

3）从 ctx.GetSlotList(true) 返回的列表中获取由 label 指定的插槽标识 slot（这里的槽可以简单理解为计算机主机上供安全硬件模块插入的槽，如 USB 插口。可能不止一个，每一个在系统内核中都有名字和标识号）。

4）尝试 10 次调用 ctx.OpenSession 打开一个会话 session（会话就是通过通信路径与安全硬件模块建立连接，可以简单理解为 pkcs11 的 chan）。

5）登录会话 ctx.Login。

6）返回 ctx、slot、会话对象 session，用于赋值给 impl 实例成员 ctx、slot，把 session

发送到 sessions 里。

关于 pkcs11，还有一点说明的，就是 SoftHSM 库，它是一个模拟硬件实现的 pkcsll，对应的系统动态库可参看 impl.go 中 FindPKCS11Lib 测试函数中所涉及的，如 Linux 下的 libsofthsm2.so。现阶段是没有安全硬件模块可以配合测试的，所以只有使用 SoftHSM 模拟测试，将 libsofthsm2.so 导入 pkcs11 对象。

5.6.4　BCCSP 工厂

对应两种 bccsp 实现，这里也有两种 bccsp 工厂：pkcs11factory.go 和 swfactory.go。Fabric 中某一模块一旦涉及工厂 factory，则说明该模块基本就是由工厂提供“窗口函数”，供其他模块调用。这里以 swfactory 为例进行讲解。

目录结构如下：

/fabric/bccsp/factory/

factory.go- 声明了默认 bccsp 实例 defaultBCCSP，bccsp 实例存储映射 bccspMap 等全局变量和这些变量的获取函数 GetXXX，定义 bccsp 工厂接口。

nopkcs11.go/pkcs11.go- 定义了两种版本的工厂选项 FactoryOpts，初始化工厂函数 InitFactories 和获取指定 bccsp 实例函数 GetBCCSPFromOpts。nopkcs11 是默认版本，可条件编译指定使用哪种版本（编译时加入 nopkcs11 或 !nopkcs11 选项）。两种版本的差异集中在是否使用 pkcs11 上。

opts.go- 定义了默认的工厂选项 DefaultOpts。

pkcs11factory.go-pkcs11 类型的 bccsp 工厂实现 PKCS11Factory。

swfactory.go-sw 类型的 bccsp 工厂实现 SWFactory。还定义了 sw 版本的 bccsp 选项。

swfactory 接口和实现如下：

```
//在factory.go中定义
//接口
type BCCSPFactory interface {

    //返回工厂的名字
    Name() string
    //返回符合工厂选项opts的bccsp实例
    Get(opts *FactoryOpts) (bccsp.BCCSP, error)
}

//在swfactory.go中定义
//实现
```

```
type SWFactory struct{}
func (f *SWFactory) Name() string {
    return SoftwareBasedFactoryName
}
func (f *SWFactory) Get(config *FactoryOpts) (bccsp.BCCSP, error) { ... }
```

实现的代码本身比较简单，Get 最终是调用的 SW 的专用生成函数 New 来生成符合 opts 的 bccsp 实例。Name 则是直接返回一个 SW 常量。

在每个 chaincode 例子中，如 /fabric/examples/e2e_cli/examples/chaincode/go/chaincode_example02/chaincode_example02.go，都使用了 chaincode 垫片 shim 中的 Start 函数。chaincode 的“垫片”shim 核心代码集中在 /fabric/core/chaincode/shim 中，该垫片所“承垫”的是与各个节点通信的任务，也即 ChaincodeSupport 服务。chaincode 形成的通信的信息，通过 shim 分发到各个节点，然后 shim 负责从各个节点收集信息，汇总返回给 chaincode，完成 chaincode 的功能。其中 shim 的 Start 函数就是用来启动一个 chaincode，定义在 /fabric/core/chaincode/shim/chaincode.go 中。在 Start 的函数中，调用了 err := factory.InitFactories(&factory.DefaultOpts) 来初始化一个默认的 bccsp 工厂。在此可以知道，这里使用的默认工厂选项（参看 opts.go），也就是使用的 swfactory。

5.7 chaincode

用户实现应用的 chaincode(application chaincode) 用 ACC 表示，系统 chaincode(system chaincode) 用 SCC 表示。对 chaincode 的元数据描述过程中，可能会涉及一些 chaincode 在具体操作时的结构上的概念，如 chaincode 的状态机、shim 端等，这些暂不用在意，与下面章节讲 chaincode 的内容对照即可。同样，按照惯例，若读者找不到所述对象、文件或函数，请自行在 chaincode 所涉及的目录中利用 grep 命令进行搜索。

chaincode 涉及的主要目录如下：

core/common
core/scc
core/chaincode
peer/chaincode
examples/chaincode/go
protos/peer/chaincode 相关的原型

5.7.1 chaincode 元数据

1. 命令行 Flag

命令行在 peer/chaincode/chaincode.go 中，是在 init() 中调用 resetFlags() 初始化了的一种 Flag，并把 Flag 的值存储在文件中定义的变量中，如 chaincodeLang，chaincodeCtorJSON，

chaincodePath，chaincodeName 等，并在 chaincode 的每个子命令初始化时，调用 attachFlags 添加子命令想要的 Flag。

2. policy

它是较复杂的 Flag 之一，也是 chaincode 中用到策略之一。chaincode 部署时需要签名，该策略是对 chaincode 的签名都还要有其他指定的签名才能生效。策略原型字符串是如 "OR(AND('A.member', 'B.member'), OR('C.admin', 'D.member'))" 这样的嵌套结构，OR(X,Y) 表示 X、Y 二者取其一，AND(X,Y) 表示 X、Y 二者都取，是组成嵌套结构的基础单位。X、Y 是如 A.member、C.admin 这样的 ORG.ROLE 格式的组织成员，ORG 表示一个组织 MSP 的 ID，ROLE 表示该 MSP 管理的成员角色（admin 或 memeber）。最终由 common/cauthdsl/policyparser.go 中的 FromString() 将策略原型字符串解析（主要使用了第三方库 govaluate）存储在一个 SignaturePolicyEnvelope（在 protos/common/policies.pb.go 中定义）对象中，该对象中的成员 SignaturePolicy 是一个递归结构，对应就可以存储嵌套结构的策略原型字符串：SignaturePolicy 的存储 SignaturePolicy_SignedBy 和 SignaturePolicy_NOutOf_ 两种类型的对象，SignaturePolicy_NOutOf_ 是递归结构的，有成员 SignaturePolicy 和成员 N，成员 SignaturePolicy 就是用于嵌套下一层策略的，成员 N 表示当前嵌套层的策略是 AND 还是 OR，1 表示 OR，2 表示 AND。最终 policy 会在调用 protos/utils/commonutils.go 中的 proto.Marshal() 之后，存储在 chaincode.go 中的变量 policyMarhsalled 中。

3. chaincodeCtorJSON

它是较复杂的 Flag 之一，指定 chaincode 要运行的函数和函数的参数，原型为 JSON 格式的字符串，如 { "Function":"func", "Args":["param"] }。可以直接调用 json.Unmarshal 将原型字符串存入一个 ChaincodeInput 对象中，作为 ChaincodeSpec 的 Input。

4. chaincode 说明书

它在 ChaincodeSpec 在 protos/peer/chaincode.pb.go 中定义，描述说明 chaincode 的结构体，简称为 CS。成员有：Type 指定 chaincode 的语言类型，当前版本的 Fabric 只支持 Go 语言的 chaincode；ChaincodeId 指定了 chaincode 的路径、名称、版本；Input 存储指定的 chaincode 的运行的函数和函数的参数。这些数据都来自于 Flag（参照上述 1 ～ 3）。

5. chaincode 部署说明书

它在 ChaincodeDeploymentSpec 在 protos/peer/chaincode.pb.go 中定义，描述说明一个 chaincode 该如何部署，简称为 CDS。成员有：ChaincodeSpec 指定了第 4 点所说的 chaincode 说明书；EffectiveDate 记录了 chaincode 何时有效的时间戳，即在 chaincode 开始运行时记录下时间点；CodePackage 存储了一个 .tar.gz 压缩包的二进制数据，这个压缩包含有 chaincode 源码以及源码所依赖第三方库；ExecEnv 标识 chaincode 运行的环境，表明是运行在 docker 容器中还是操作系统之中。其中 CodePackage 经 core/container/vm.go 中的

GetChaincodePackageBytes()，选择 go 平台的 goPlatform 对象后，最终调用 core/chaincode/platforms/golang/platform.go 中的 GetDeploymentPayload，根据 chaincode 说明书中记录的 chaincode 信息，生成压缩包。这个过程的复杂之处主要在于收集 chaincode 源码所依赖的第三方库（这些依赖的第三方库应该放在 GOPATH/src 下），在搜集过程中，排除了 GOROOT 和 Fabric 项目已经提供的库，也排除了 chaincode 源码目录路径中所有的 vendor 目录中的库，剩下的所依赖的第三方库都被重新映射到 chaincode 源码目录下的 vendor 目录。最终，将源码文件和所依赖的库都存储在一个 .tar.gz 压缩包中返回。如 chaincode 源码的路径是 GOPATH/src/hyperledger/fabric/examples/chaincode/go/chaincode_example02/，假设该 chaincode 源码依赖第三方库 github.com/jmoiron/sqlx（放在 GOPATH/src/ 下，实际上不依赖），则生成的压缩包中的路径为 src/hyperledger/fabric/examples/chaincode/go/chaincode_example02/，chaincode_example02 目录下包含 chaincode 所有源码和一个 vendor 目录，vendor 目录中包含路径 github.com/jmoiron/sqlx/，sqlx 目录中包含 sqlx 库全部的文件。

6. ccpackfile 包文件

它是一种由 chaincode 命令的 package 或 signpackage 子命令生成的 chaincode 二进制部署包。前者由 CDS 对象生成，后者由 Envelope 对象（包含签名过的 CDS）生成。将这两者形成的 ccpackfile 使用 ioutil.ReadFile 读入一个 buf 中后，可以先使用 CDSPackage 对象或 SignedCDSPackage 对象的 InitFromBuffer 将 buf 中的数据转入对象，然后调用对象的 GetPackageObject 从对象中抽取部署数据，将其尝试转化为 CDS 或 Envelope 供部署时使用。我们将 CDS 和用于部署的 Envelope 都称作部署数据。

7. CDSPackage/SignedCDSPackage 对象

它在 core/common/ccprovider/ 下定义，也是一种存储 chaincode 部署说明书数据的对象，但同时也提供了一系列接口供各种情况下调用。所以它更像是扮演了一种存储 CDS 数据的中间角色，把 CDS 数据转化为其他所需的格式。

8. ChaincodeInvocationSpec

它是 chaincode 调用说明书，简称为 CIS，主要存储了一份 chaincode 说明书，只不过这份说明书可能是关于某一个系统链码的说明书。Input 中存储的函数变为 install/upgrade 等对 ACC 进行操作的函数，而用户的部署数据（CDS 或 Envelope）则变为该函数的参数。比如用户要安装一个自己的 chaincode，生成的调用说明书中会使用一份系统链 lscc 的说明书，而对于 lscc 说明书中的输入 Input 来说，操作的函数为 install，函数的参数为用户 chaincode 的部署数据。

9. Proposal/SignedProposal

Proposal 封装了 chaincode 的数据，如部署包等，作为一个申请消息，让节点签名后，会同签名数据一同放入 SignedProposal。而 SignedProposal 就是 chaincode 可以通过 grpc 与

节点进行通信的数据包结构，即 ACC 向节点发送执行交易的消息，都是以 SignedProposal 消息的形式发送出去的。

10. CCContext

它是就是 chaincode context，即链码内容，也是描述 chaincode 自身信息的一种载体，记录了 chaincode 的一些关键信息，如成员 ChainID 指定了所代表的 chaincode 所在的链的 ID，成员 Name 指定了所代表的 chaincode 的名字，成员 Syscc 指定了所代表的 chaincode 是否是 SCC。

11. transactionContext

它是交易上下文，在交易中产生，以交易 id 为 key 记录在 Handler 的一个 map 中，随用随删。比如一个部署交易中，在 Handler 处理这个交易期间，会产生一个这样的交易上下文，存储关于这个交易的一些信息，供处理交易的主体使用。在该次部署交易完成后，则会将其删除。

12. Chaincode Message

ChaincodeMessage 是链对象的服务端和 shim 端进行消息交互的主要的消息载体，在 protos/peer/chaincode_shim.pb.go 中定义。成员 Type 指定了消息的类型，有 ChaincodeMessage_REGISTER、ChaincodeMessage_INIT 等。成员 Txid 指定了该消息所在的交易编号（每个 chaincode 执行的交易都会有一个编号，每个交易会使用多个类型的 ChaincodeMessage 在 chaincode 的服务端和 shim 端进行交互）。成员 Timestamp 是时间戳；成员 Payload、Proposal 是消息承载的 chaincode 数据，如源码包、执行的参数等（只要是 []byte 格式的，那其承载的数据就自由得多），可能有，也可能是空；成员 ChaincodeEvent 是 chaincode 在 Init 或 Invoke 时给回的所要触发的事件，比如，chaincode 部署的时候，部署完毕后，shim 可能给回服务端一个 Event 让服务端去触发，做一些必要的其他事情。同样，可能有，也可能是空。

13. CCPackage/SignedCDSPackage

它们分别定义在 core/common/ccprovider/ 下的 cdspackage.go 和 sigcdspackage.go 中，两者都实现 ccprovider.go 中定义的 CCPackage 接口，可以从一个 Marshal 过的 CDS 或 Envelope 抽取出其中所承载的关于 ACC 的数据来初始化自身，以方便 SCC 检查 ACC 的数据是否符合要求，最终也是以这种两类结构写入文件系统以保存 ACC 的。

14. ChaincodeData

它是承载 ACC 数据的，用于 ACC 在系统中部署时，最终向账本提交安装的数据形式。

15. StartImageReq

它是容器相关的数据结构，在 core/container/controller.go 中定义。包含了部署一个 chaincode 时启动一个容器所需要知道的如环境变量、网络 ID、节点 ID、部署包、建立函

数、执行函数等数据。相应的，有 StopImageReq、DestroyImageReq 两个结构体。

16. ChaincodeStub

它是相当重要的一个结构体，接口定义在 core/chaincode/shim/interfaces.go，实现在 chaincode.go 中。但是这个可以自己实现，参见 core/chaincode/shim/mockstub.go 和 mockstub_test.go。这个结构体是链码执行 Init、Invoke 两个接口时直接使用的数据，即一个交易到底要做什么的信息都会封装到这个结构里。

5.7.2 chaincode 元工具

chaincode 元工具包括以下内容：

1. Signer

它是签名者，由 peer/chaincode/common.go 中的 InitCmdFactory 初始化，使用的是一个节点的本地 MSP 服务对象 bccspmsp 中的 signer，即一个 SigningIdentity，可参看《Fabric 源码分析 12》。签名者的作用就是调用 CreateInstallProposalFromCDS 和 GetSignedProposal，对 chaincode 的部署数据签名（CDS 或 Envelope），形成一个已签名的申请 SignedProposal。

2. EndorserClient

它是背书客户端，由 peer/chaincode/common.go 中的 InitCmdFactory 初始化。在《Fabric 源码分析 11》中所提及的 Endorser 服务，所述的即为背书的服务端，该服务端会随着一个节点的 start.go 的 serve 函数运行起来。背书客户端的作用是调用 ProcessProposal，将签名者生成的 SignedProposal 发送给背书服务端执行交易后背书，获得一个申请应答 ProposalResponse。ACC 的交易都是通过它来发起的。

3. BroadcastClient

它是一个连接 orderer 服务的广播客户端。chaincode 用其来发送交易的结果数据给 orderer 节点，供其进一步处理。

4. ccprovider

它是 chaincode 提供者，接口在 core/common/ccprovider/ccprovider.go 中的 ChaincodeProvider，唯一的实现是 core/chaincode/ccproviderimpl.go 中的 ccProviderImpl（ccprovider.go 中初始化的也是这个实例，参看 ccproviderimpl.go 的 init()）。这个工具相当于一个封装层，置于 chaincode 与各种结构的 chaincode 数据之间。比较明显的就是这个接口的方法中，很多参数都是 interface{} 类型。即，各种不同目的各种形式的 chaincode 数据要经过它，去往各个适合其该去的地方去做事情。

5. ChaincodeSupport

它是 chaincode 支持者，是一个全局单例，在 core/chaincode/chaincode_support.go 中实现和定义。为整个 chaincode 的框架提供支持，更精确地说，是提供 chaincode 整个服务端

的框架支持。相当于一个大杂烩，关于 chaincode 用到的、需要的、涉及的，都可能在这里存储并引导执行 chaincode 的动作。但它也仅仅是这样，并不真正掺和 chaincode 的执行，也就是说，chaincode 需要的，ChaincodeSupport 尽量满足，比如一个账本对象。但之后处理或账本对象自身如何处理交易，它就不管了。再比如，要部署时，它交给属于 Chaincode 的 Handler 之后也不管了就等结果了，再比如，需要何种类型的容器，它根据 Chaincode 的需要返回给 Chaincode inproc 容器或 Docker 容器对象，然后让容器对象自己跟 Chaincode 接触和交互，它也不管。

6. container

它是容器，即 chaincode 最终要寄居的地方，分有两种：inproc 容器和 docker 容器。

7. Handler

它分为服务端的 Handler 和客户端的 Handler，各自封装了一个 FSM 状态机，chaincode 的交易就是由这两个 Handler 的状态机驱动的，相当于 chaincode 交易的控制器和发动机。

8. txsimulator

它是交易模拟器，模拟交易发生的事情，其实就是在内存（典型的就是 map）中记录交易产生的数据。在一个 chaincode 交易进行后，会将数据由交易模拟器记录一份，然后再把数据交给 orderer 服务处理（最终提交至账本）。类似的还有 historyQueryExecutor 之类的查询工具。

5.7.3　SCC 的注册和部署

SCC 的注册和部署由于集成于 peer 内部，直接与用户交互之处甚少，因此也最简单。在此将注册和部署合为一节。首先要理解的是，SCC 和 ACC 本质是一样的，但 ACC 是处理用户的数据并根据用户指定的操作进行执行的，而 SCC 是处理 ACC 自身的，也就是说，对于 SCC 来说，ACC 就是它要处理的数据。SCC 具体负责 ACC 的安装、部署等，就算 ACC 要查个数据，也要经过 SCC，让 SCC 去查，查出来后由 SCC 给 ACC。之所以区分 SCC 和 ACC，按照官方文档的解释就是，这样做可以使项目源码的弹性十足。先讲 SCC 是最合适的，一是 SCC 比 ACC 简单；二是 SCC 的机制也是 ACC 的机制，SCC 代码走的线路，基本也是 ACC 代码要走的线路，熟悉了 SCC，再了解 ACC 要顺利得多。很自然，由于 ACC 要与开发者和使用者进行交互，因此要更复杂一些。

1. 概述

SCC 的 Register 相当于 ACC 的 Install，SCC 的 Deploy 相当于 ACC 的 instantiate。这里和之后讲 ACC，统一用 Install 和 Deploy 来描述两个动作。只有这两个动作执行完毕，一个 chaincode 才算真正可以使用。

SCC 启动占用的是 inproc 容器，可以当作内存中概念上的容器，在 core/container/

inproccontroller 下实现；ACC 启动占用的是 docker 容器，在 core/container/dockercontroller 下实现。

chaincode 接口是定义在 core/chaincode/shim/interfaces.go 中的 Chaincode，只有两个接口：Init(stub) 和 Invoke(stub)，统一用同文件中的 ChaincodeStubInterface 接口实例作为唯一的参数。ChaincodeStubInterface 接口唯一的实现是在同目录下的 chaincode.go 中的 ChaincodeStub。

每一个 chaincode 都会执行两个 Handler，每一个 Handler 都是一个以状态机（FSM）驱动的通信机器，在驱动的过程中执行具体的状态事件，完成一个 chaincode 所需做的事情。一个可以称为服务端 Handler（相当于服务端），在 core/chaincode/handler.go 中实现；另一个可以称为 shim（shim 本身有垫片的意思，可以理解为项目源码与开发者之间贴合的垫片）端 Handler（相当于客户端），在 core/chaincode/shim/handler.go 中实现。后续文中统一以 ServerHandler 和 ShimHandler 区分。对状态机，即第三方库 github.com/looplab/fsm 的概念和操作不太熟悉的读者，建议学习一下相关内容（可参考《Fabirc 源码解析 7》），这是能看懂这部分源码的前提。若粗略分一下的话，core/chaincode 下的为 chaincode 服务端的代码，主要用于处理 chaincode 的请求；core/chaincode/shim 下的为 chaincode 客户端的代码，用于定义供开发者使用的接口和客户端提交申请。

一个 chaincode 实质上还是一个结构体对象，该结构体实现了 Chaincode 接口。SCC，如 core/scc/lscc/lscc.go 中的 LifeCycleSysCC，ACC，如 examples/chaincode/go/chaincode_example01/chaincode_example01.go 中的 SimpleChaincode。

所有的 SCC 对象和原始数据定义在 importsysccs.go 中，下面仅以 lscc 这个 SCC 作为例子进行分析，lscc 的 chaincode 对象 LifeCycleSysCC 具体实现在 core/scc/lscc 中。因 chaincode 自身已经比较复杂，因此与 chaincode 相关却不属于 chaincode 主题的对象，下面将简单略过。开头部分尽量讲详细，后边与前文重叠的部分将不再详述，如两个 Handler 之间的数据流转，第一次涉及会比较详细地讲，之后就将简略述之。

2. 安装

在 peer/node/start.go 的 serve 函数中，调用了 registerChaincodeSupport，这其中执行了 scc.RegisterSysCCs()，针对每一个 SCC 依次执行 core/scc/sysccapi.go 中实现的 RegisterSysCC(syscc)，进行注册（安装），自然也包括 lscc。

if !syscc.Enabled || !isWhitelisted(syscc) 可判断 lscc 是使能的且处于白名单之中，即 lscc 的 Enabled 要为 true，且在配置文件 core.yaml 的 chaincode.system 项中，lscc 要配置为 enable（或 yes/true）。如此，lscc 的注册才能继续。

inproccontroller.Register(lscc.Path,lscc.Chaincode) 在 core/container/inproccontroller 定义、注册（即安装）lscc。前面说过，SCC 启动占用的是 inproc 容器，这里 Register 所做的就是把 lscc 的 Chaincode 成员（lscc 实际的 chaincode 对象 LifeCycleSysCC）包装进一个

inprocContainer 容器，然后以 lscc.Path 为 key 放进 typeRegistry 这个 map 中，这就算注册（安装）完毕了。从这一点就可以体会，相较于 ACC 的安装启动一个 docker 容器，然后将 ACC 数据放入 docker 容器内指定的目录，inproc 容器只是一个叫法上的容器。typeRegistry 这个 map 的作用也仅仅是存储注册的 SCC。

3. 部署

在 peer/node/start.go 的 serve 函数中，调用了 initSysCCs()，这其中执行了 scc.DeploySysCCs("")，针对每一个 SCC 依次执行 core/scc/sysccapi.go 中实现的 deploySysCC("", syscc)，进行部署。这里需要留意第一个参数，是空 ""，这个值是赋值给 chainID 的。chainID 的值对之后每一个 chaincode 的一系列操作有很大的影响，自然，也包括 lscc。这里提前说一下，与 chainID 一样，之后会遇到 channelID、channel、chain 这样的字眼，其实指的都是指链的概念。一条链有一套属于自己的管理范围、工具、账本等，只有一个 chaincode 属于这条链，才能使用这条链的工具、账本，并受其管理，这个 chaincode 才能在链上起作用。SCC 不属于任何一个链，它属于 peer 节点，也有权限，它只是处理 ACC 的请求，ACC 在请求中提供 chainID，SCC 根据这个 chainID 从相应链上取出 ACC 所需要的工具或数据返回给 ACC 使用即可。

if !syscc.Enabled || !isWhitelisted(syscc)，如同安装一样，检测 lscc 是否使能且处于白名单之中。

ctxt := context.Background()，if chainID != "" {...} 声明了一个 Context，这也是一个之后一直都在使用的变量（对于 Context，不熟的读者需要自行学习了）ctxt。然后 if 判断 chainID 是否为空，若不是空的，if 分支中所做的事情就是查看一下这条链上的账本和账本交易模拟器是否正常。换句话说，若 chainID 不为空，即指定了 lscc 部署到哪条链上，那肯定是之后部署 lscc 的时候会用到这条链的 Ledger 和 Ledger 的 TxSimulator（否则不会为了不做无用功而提前检查）。这也是一点可说的，算是一个小技巧：在追溯比较冗长的代码过程中，前期函数中使用的 if 判断，尤其是函数中开头处的 if 判断，也有很好的间接提示作用，在后期的函数中可以做到相互印证，来判断自己追溯路线的是否正确。比如这里的 if，若 chainID 不为空，后期追溯的过程中我们发现并没有使用 Ledger 和 TxSimulator，那基本就可以判断你追溯错了。这一小技巧在后面还会提及。当然，这里我们还是遵循尽量实例化的原则，chainID 为空，因此该 if 分支不会被执行。

spec := &pb.ChaincodeSpec{...}，chaincodeDeploymentSpec, err := buildSysCC(ctxt, spec)，cccid := ccprov.GetCCContext(...)，它们根据原始的 lscc 数据，一路封装最后得到 lscc 的部署包 CDS、CCContext，并将这些数据连同 ctxt 传入下一步准备执行部署。

ccprov.ExecuteWithErrorFilter(...) 是 chaincode 执行交易的一个路线之一（另一个路线是 core/chaincode/chaincodeexec.go 中的 ExecuteChaincode()，ACC 走的是这条线），经 ccprovider 导航，执行调用到 core/chaincode/exectransaction.go 中的 ExecuteWithErrorFilter(...)，lscc 的数据也跟着到此，进而直接调用通用文件中的 Execute(...)，部署动作算是正式开始。需要说

仅供非商业用途或交流学习使用

明的是，不单是 lscc 的部署、所有的交易，无论是 ACC 的安装部署、查询，还是 Invoke 等，最终都会从 Execute(...) 开始，这里算是交易产生和最终结束的地方。遵循实例化原则，不会进入执行的 if 分支则不讨论。

Execute(...) 接收的是 lscc 数据，利用这些数据，主要执行了两步：theChaincodeSupport.Launch() 和 theChaincodeSupport.Execute()。在 Execute 执行完后，会对交易返回的应答消息 resp 进行判断，若成功，将返回 resp 和 resp 中的“倒钩事件”。下面将分别讲述 Launch 和 Execute。

（1）Launch

首先看 Launch，根据 lscc 封装的数据，cds 不为空，cID = cds.ChaincodeSpec.ChaincodeId，cMsg = cds.ChaincodeSpec.Input，canName := cccid.GetCanonicalName()，从封装的数据中抽取出目前需要的数据。首先调用 chaincodeSupport.chaincodeHasBeenLaunched(canName) 进行 if 判断，查看 lscc 是否已经被执行过。这个 if 判断已经可以看出 Launch 这个函数最终要做的事情了：将 lscc 以 canName（就是 chaincode 名字 +’:’+ 版本号，即 lscc:1.0.0，这算是 chaincode 内部认可的权威名字）为 key，将 lscc 对应的 ServerHandler 放入这个 map 中。这一点还是上文所说的技巧之一。当然，这里 lscc 是第一次部署，其数据不会存在 chaincodeMap 中，所返回的 chrte 自然为空。

if cds == nil { ... }，该分支不会进入，但是在 ACC 安装部署时会进入，这里不讨论。

if (!....userRunsCC || cds.ExecEnv == ..) && (...) { ... }，对照 lscc 封装的数据会发现，该分支可以进入。分支中，判断 if cds.CodePackage == nil 会进入，但是 if !(....userRunsCC || cds.ExecEnv == ....SYSTEM) 不会进入。直接到了 builder := func() ...，定义了建立 Docker 容器的对象 builder，并将 Builder 作为参数之一传入 chaincodeSupport.launchAndWaitForRegister(...)，真正开始 Launch 的工作。注意，这个 builder 对于 SCC 来说，在之后不会使用，对于 ACC 来说才会用到。

从 launchAndWaitForRegister(...) 名字可以看出，它是一个等待函数，即一直要等到 Register 完毕才会返回。这里的 Register 是指 lscc 服务端的 Handler 的状态，即要等到 lscc 的 ServerHandler 处于 REGISTER 状态才会返回，对应该函数结尾处的 select-case 等待，顺理成章。在函数的开头，再次调用了 chaincodeSupport.chaincodeHasBeenLaunched(canName)，查看 lscc 是否已经被执行过。然后开始准备数据，形成 ipCtxt、vmtype、sir（包含 CCID、Env、Args 等数据），传入 container.VMCProcess()（core/container/controller.go 中定义），在其执行成功的情况下进入等待。

（2）Execute

将目光重新定位到 core/chaincode/exectransaction.go 中的 Execute(...) 函数，在执行完毕 theChaincodeSupport.Launch(...) 后，继续。ccMsg, err = createCCMessage(...) 根据 Launch 返回的 CDS.CS.ChaincodeInput（实际为空）和之前的 cctyp 生成一个 ChaincodeMessage_INIT 类型的 ChaincodeMessage 传入 theChaincodeSupport.Execute(...)（下文若非特指，凡提到的

仅供非商业用途或交流学习使用

Execute 函数均指此函数）。

在此罗列一下进入 Execute 的参数：ctxt 和 cccid 仍是 lscc 部署之初生成的，ccMsg 是当前生成的，executetimeout 是 ChaincodeSupport 设置的超时时间。

if chrte, ok := chaincodeSupport.chaincodeHasBeenLaunched(canName)，再次检查 lscc:1.0.0 是否被 Launch 过，同时若已被 Launch 也返回了 lscc 对应的 ServerHandler。到这一步，lscc:1.0.0 必定已被 Launch。

notfy, err = chrte.handler.sendExecuteMessage(...)，利用 lscc 的 ServerHandler，目的在于运行 ChainMessage 中所携带的数据指向的动作，并返回通知通道给 notfy。当顺利返回后，Execute(...) 函数将进入常规的 select-case 等待（由此可知 sendExecuteMessage 中的具体的任务依旧是异步执行）。

在 sendExecuteMessage 函数中，handler.createTxContext(...)，依然用 txid 为 key 创建了一个交易上下文对象 transactionContext（这个对象会在 Execute 结束前调用 chrte.handler.deleteTxContext(msg.Txid) 被删除），然后调用 handler.triggerNextState(msg, true) 将 ChaincodeMessage_INIT 类型的消息发送到 handler.nextState 通道，再将交易上下文对象的 responseNotifier 通道返回给 Execute 函数的 notfy，并让 Execute 函数进入等待。

ServerHandler 对于 handler.nextState 通道来的 ChaincodeMessage_INIT 类型消息不会触发任何状态的改变和事件函数，只会把该消息原封不动地发给 ShimHandler。ShimHandler 收到此消息后，将触发 beforeInit 事件函数。beforeInit 事件函数中实际执行任务的又是 handleInit(msg) 函数。

handleInit(msg) 函数也是一个比较典型的处理流程（handleTransaction 函数也是这种流程，ACC 中会讲到）。函数整体只新起一个 goroutine，在 goroutine 中，在 defer 中执行最后要发送的消息 nextStateMsg（可能是正常的消息，可能是错误消息），然后定义了一个 errFun 函数，一旦检测有错，则返回一个 ChaincodeMessage_ERROR 类型的 ChaincodeMessage 消息（即错误消息）给 nextStateMsg，随即 return，也就触发了 defer。

4. 部署后的状态

theChaincodeSupport 中的 runningChaincodes.chaincodeMap 中存在以 lscc:1.0.0 为 key 的 lscc 的 ServerHandler 实例。lscc 的 ServerHandler 和 ShimHandler 均处于 ready 状态，且接收发送消息的 goroutine 都在运行当中。其他的 SCC 均与 lscc 状态相同。

5. 遗留问题

关于版本数据问题，笔者还未搞清楚，但无论对于 SCC 还是 ACC 来说，这都是个比较关键的数据，交易的一路都在使用。SCC 所用到的版本数据是在 deploySysCC 中组装 CCContext 时，调用 version := util.GetSysCCVersion() 得到的，这个版本数据来自于 common/metadata/metadata.go 中的 Version 变量。然而这个变量整个项目中没有找到给其赋值的地方，按照注释的说法是来自于 Fabric 的 Makefile 文件中的 GO_LDFlAGS，查看 Makefile 文件确有版本号相关的数据，如 BASE_VERSION = 1.0.0。但是笔者对 Makefile

仅供非商业用途或交流学习使用

不精通，不清楚在编译项目时是如何将 Makefile 文件中的版本数据赋值给 metadata.go 中的 Version 变量的。若精通 Makefile 文件的读者也在研究 Fabric，法不吝赐教。

5.7.4 ACC 的安装和部署

peer chaincode install 命令执行安装操作，命令定义在 peer/chaincode/install.go 中，这也是安装的起点。另外需要注意的一点是，这个命令是在 peer node start、peer channel create、peer channel join 命令依次执行完毕之后所执行的，即执行 install 之时，peer 节点的基本的模块（包括 SCC）都已初始化完毕，channel 也已经建立。

根据实例化的原则，从 install_test.go 中提取了一句实际的 install 命令：

```
peer chaincode install -n example02 -p github.com/hyperledger/fabric/examples/chaincode/go/chaincode_example02 -v anotherversion
```

本节将以此句命令为例子。安装的 ACC 是 example02，存在于 -p 指定的路径下，版本是 anotherversion。-l 没有给定，自然取默认的值，为 Go 语言。

install 安装使用的 chaincode 数据有两种形式，一种是 CDS，一种是 chaincode package/signpackage 命令形成的 ccpackfile 包。因为我们没有涉及过这两个命令，因此这里只以前一种 CDS 数据包为例，叙述安装过程。

install 能够识别的 flag 有 -l、-c、-p、-n、-v。其中 -c 不常用，其指定的是 ACC 所要执行的函数和函数的参数，一般在具体执行的时候，如查询或转账的时候再给定。

install 最终所要做的事情，就是将 example02 的源码包放入 docker 容器的安装目录中。

1. 生成签名申请包

以 peer/chaincode/install.go 的 chaincodeInstall(...) 为起点（设定 ccpackfile==""），说明以下几点：

1）chainID 在 SignedProposal 中为空，即安装所要做的仅仅是把 example02 的压缩包放入容器的指定目录中而已，只有部署的时候才需要指定部署到哪条链上。

2）CDS 的 CodePackage 是 example02 的代码压缩包，包括源码和依赖的第三方库。因为 SCC 的一切都在项目编译时编进 peer 中了，所以 SCC 的 CDS 的这个字段就是空的，而 ACC 需要安装源码，所以这个字段在正常的情况下肯定不是空的。这个压缩包最终是在 core/chaincode/platforms/golang/platform.go 中的 GetDeploymentPayload() 打包的。该程序的复杂之处在于除了要打包 example02 源码，以及打包 example02 直接依赖的但 Go 标准库未提供的第三方库，还要打包这些第三方库直接依赖但 Go 标准库未提供的第三方库（即 example02 间接依赖但 Go 标准库未提供第三方库）。这么做的目的就是让一个 chaincode 无论放到哪个容器里，不会因为缺少某个第三方库而编译失败。同时这个打包的过程也就要求我们在执行 example02 安装的时候，要实现把其依赖的第三方库事先放到 GOPATH/src 下。

仅供非商业用途或交流学习使用

3）txid 是在 /protos/utils/proputils.go 中的 CreateChaincodeProposalWithTransient(...) 中计算出来的，是哈希（peer 节点的 MSPID+ 证书元数据 + 随机数 nonce）的值，可以就把它当作一个唯一的字符串，不用太过深究。

2. 处理安装申请

Endorser 服务端在 core/endorser/endorser.go 中的 ProcessProposal(...) 处，接收到来自 Endorser 客户端发送的 SignedProposal 和一个调用之后一路都会用到的 Context 上下文 ctxt。

在 ProcessProposal(...) 中，从开始至 var txsim ledger.TxSimulator 处，之上的代码全部是一边解压抽取 SignedProposal 中的数据，一边验证这些数据。

var txsim ledger.TxSimulator 和 var historyQueryExecutor ...，一个是交易模拟工具，一个是历史查询执行工具。由于是 Install，且 chainID 为空，这两个值在之后都一直为空。if chainID != "" 的分支也不会进入（当部署 example02 时，chainID 不为空，则会进入此分支，根据 chainID 获取这两个工具，供之后部署使用）。

ProcessProposal(...) 所做的主要两件事如下：

1）e.simulateProposal(...)，模拟执行申请。

2）e.endorseProposal(...)，背书申请执行的结果。但是由于 chainID 为空，所以 install 命令不会执行此步（同样，部署时会用到），而是直接以 ProposalResponse 的形式返回 1）中执行的结果。

3. 执行申请

在此罗列一下传入 e.simulateProposal(...) 的参数：ctx 为上文所述的上下文 ctxt（至此未有更新）；chainID 为空；txid 为交易 ID；signedProp 是客户端发送来的原数据；prop 是从 signedProp 中抽取出来的 Proposal；hdrExt.ChaincodeId 也是抽取出来的数据，只包含一个值为 lscc 的 Name 字段；txsim 为空。

在 e.simulateProposal(...) 中，前期又做了些简单的抽取和检查的事情：cis, err := putils.GetChaincodeInvocationSpec(prop) 从 prop 中抽取出 CIS。if err = e.disableJavaCCInst(cid, cis); err != nil 通过判断 example02 的 CDS.CS.Type 来断定要安装的是否为 Java 源码，关于这点注释说得很清楚，当前版本不支持 Java 写的 chaincode，但这部分在将来会被移除。if e.checkEsccAndVscc(prop); err != nil，escc 和 vscc 对 prop 的检查，但是当前版本未做什么实际的检查，以后的版本可能会加入。if !syscc.IsSysCC(cid.Name){...}else{ version = util.GetSysCCVersion() }，由于 cid.Name 就是 lscc，因此只会进入 else 分支，得到的为 lscc 的版本值为 1.0.0。

最后一步调用了 e.callChaincode(...)，执行 CIS 指定的动作。上文所述的另一个函数 endorseProposal() 最后也是调用这个函数开始执行申请的任务的。也就是说，e.callChaincode(...) 能实现什么效果，做什么事情，完全是由传入的参数决定的，这个函数可以算是实际开始执行申请的起点。

仅供非商业用途或交流学习使用

在此罗列一下进入 e.callChaincode(...) 的参数：ctx 依旧为 ctxt；chainID 为空；version = 1.0.0；txid 为交易 ID；signedProp/prop/cid/txsim 不变；callChaincode() 函数做了 3 件事：

1）cccid := ccprovider.NewCCContext(...)，根据传入的参数，创建一个 CCContext 对象供执行申请所用。

2）chaincode.ExecuteChaincode(...)，执行申请。

3）if cid.Name == "lscc"&& len(cis.ChaincodeSpec.Input.Args) >= 3 && ...，这一步只有部署，升级的交易才会进入此分支。分开讲解两个函数。

（1）Launch

Launch(...) 函数在 ACC 安装的情况下执行不了太久，canName := cccid.GetCanonicalName() 等到的 canName=lscc:1.0.0，此值会是程序进入 if chrte, ok = chaincodeSupport.chaincodeHasBeenLaunched(canName); ok 分支，进而进入 if chrte.handler.isRunning() 分支而返回。因为负责处理安装 example02 的 lscc 已经 Launch 过了，所以这样安排顺理成章。

（2）Execute

在此罗列一下传入 Execute(...) 函数的参数：ctxt 依旧未更新；cccid 即 CCContext；ccMsg 即 ChaincodeMessage；executetimeout，为超时时间。

Execute(...) 函数中，canName := cccid.GetCanonicalName() 再次得到 lscc:1.0.0，然后通过 chrte, ok := chaincodeSupport.chaincodeHasBeenLaunched(canName)，获取了已部署过的 ServerHandler，再 chrte.handler.sendExecuteMessage(...) 开始使用该 ServerHandler 以触发状态机进入运行，最后进入 select-case 等待。

在此罗列一下传入 sendExecuteMessage(...) 的参数：ctxt 依旧未更新；cccid.ChainID 为空；msg 即 ChaincodeMessage；cccid.SignedProposal 即 SignedProposal；cccid.Proposal 即 Proposal。

在 sendExecuteMessage(...) 中，通过调用 handler.triggerNextState(msg, true)，ServerHandler 将 msg 发给自身的 handler.nextState 通道，以触发 ServerHandler 的状态机进入下一个状态。

ServerHandler 的 processStream() 收到来自 handler.nextState 通道的 msg，先交给 handler.HandleMessage(in) 处理，ServerHandler 状态机无任何变化，然后 handler.serialSendAsync(in, errc) 给 lscc 的 ShimHandler 发送 msg。

lscc 的 ShimHandler 的 chatWithPeer() 收到 msg，交由 handler.handleMessage(in) 处理，触发 beforeTransaction 事件函数。该事件函数主要调用同文件中的 handleTransaction(...) 函数。

handleTransaction(...) 函数主要做的就是根据 ShimHandler 收到的 msg 生成并初始化一个 ChaincodeStub，然后 handler.cc.Invoke(stub) 调用 lscc 的 Invoke() 方法对 example02 进行安装。

在 lscc 的 Invoke() 中，args := stub.GetArgs() 获取到的是 CIS.CS.Input.Args。这个数组的值来自于 protos/utils/proputils.go 中 createProposalFromCDS() 中的 case "install": 中的

仅供非商业用途或交流学习使用

ccinp。因此 Invoke() 中，function := string(args[0]) 得到 function 的值是 "install"，switch function 会进入 case INSTALL: 的分支。

case INSTALL: 分支中，首先，lscc.policyChecker.CheckPolicyNoChannel(...) 专门使用了检测未指定 Channel 的 chaincode 的函数来检查要安装的 example02，这也侧面验证了上文生成签名申请包章节中对 chainID 的描述。然后，depSpec := args[1] 取出来的就是 example02 的 CDS，不过此时的 CDS 仍是被 Marshal 过的。最后，lscc.executeInstall(stub, depSpec)，调用 lscc 的函数，依据 stub 和 example02 的 CDS，执行安装。

4. 安装后的状态

peer 节点的 chaincodeInstallPath 目录下，会有一个名为 example02.anotherversion 的文件，该文件即为 example02 源码压缩包。

5. 部署概述

peer chaincode instantiate 命令执行部署命令，命令定义在 peer/chaincode/instantiate.go 中，这也是部署的起点。另外需要注意的一点是，这个命令是在 peer node start、peer channel create、peer channel join、peer chaincode install 命令依次执行完毕之后所执行的，即执行 instantiate 之时，peer 节点的基本的模块（包括 SCC）都已初始化完毕，channel 已经建立，ACC 也已安装。

根据实例化的原则，从 instantiate_test.go 中和官方文档中提取并整合了（尽量能用的 flag 都用上）一句实际的 instantiate 命令：

```
peer chaincode instantiate -n example02 -v anotherversion -o orderer.example.com:7050 -C testchain -c '{"Args":["init","a", "100", "b","200"]}' -P "OR ('Org1MSP.member','Org2MSP.member')"
```

本处将以此句命令为例子。-n 指定部署的 ACC 是 example02，-v 指定版本是 anotherversion，-o 指定连接的 orderer 服务实例的端点是 orderer.example.com:7050，-C 指定要部署的链是 testchain，-c 指定执行的函数和函数的参数，-P 指定策略。

同样，本文将重点放在不同于 install 之处。

instantiate 能够识别的 flag 有 -l，-c，-n，-v，-P，-E，-V，-C。相对于 install，这里就指定了 -c，要求部署的时候初始化 a、b 两个账户。

ACC 的部署是存储在 docker 容器中的，所启动的两端 Handler 通过 grpc 进行通信。

instantiate 最终要做 3 件事情，涉及 3 条链码：

1）lscc 执行部署交易将 example02 的源码放入自己的写集。

2）example 执行自身的部署交易，启动 example02 容器并与 peer 节点通过 grpc 通信进行初始化，最后将初始化的状态写入自己的写集。

3）获取 lscc，example02 的读写集，使用 escc 进行背书，然后将签名、读写集等部署产生的数据封装成 Envelope，发送到 orderer 节点交由其处理。

仅供非商业用途或交流学习使用

6. 部署

起点在 peer/chaincode/instantiate.go 中的 chaincodeDeploy(...)，主要做了两件事：

1）env, err := instantiate(cmd, cf)，与 install 命令执行的线路类似，对 example02 进行部署，并返回部署结果 env。

2）cf.BroadcastClient.Send(env)，在部署成功的前提下，向各个节点广播部署结果 env。下文将分别详述。

instantiate() 是部署的起点，同 install 一样，从最原始的命令行数据开始，一路组装数据，至 SignedProposal。

这里需要注意的是，中途组装的 example02 的 CDS 中，CodePackage 为 nil(这个很好理解，install 已经将 example02 的源码放入指定目录了，部署的时候自然就不必再携带 example02 的源码数据)。chainID 值为 testchain，不再为空，即要把 example02 安装在 testchain 上。CIS.CS.Input.Args 的值依旧来自于 protos/utils/proputils.go 的 createProposalFromCDS() 中的 ccinp，不过不同于 install，这次进入的是 case "upgrade": 分支（case "deploy" 中执行了 fallthrough），这点将直接影响 lscc 的 Invoke 所进入的分支。

cf.EndorserClient.ProcessProposal(...) 将组装好的 SignedProposal，连同一个之后一路在用的 Context 上下文 ctxt，一起发给 Endorser 服务端。

Endorser 服务端在 core/endorser/endorser.go 中的 ProcessProposal(...) 接收到来自客户端的 ctxt 和 SignedProposal。ProcessProposal(...) 所做的事情依旧如 install 时所描述的那样，但不同与 install 之处在于，chainID := chdr.ChannelId 获取的值为 testchain。

在 e.simulateProposal(...) 中，不同于 install 所述之处在于，执行 e.callChaincode(...) 之后，会进入 if txsim != nil 分支，也会进入最后的 if cid.Name == "lscc"&& ... 分支。进入 callChaincode(...) 的参数，除了新抽取出来的 cis 以外，其余均未变。

在 callChaincode(...) 中，会进入 if txsim != nil 分支。一直到调用 lscc 的 Invoke（core/lscc/lscc.go 中），args := stub.GetArgs() 获取得到的是 ccinp，function := string(args[0]) 得到的值是 deploy，因而之后的 switch-case 会进入 case DEPLOY: 分支。在 case DEPLOY: 分支中，依次对 args 的每个值进行了检查，对以下参数进行了修补，如 escc/vscc 若为空，则给默认值。最后调用 lscc.executeDeploy(...)，开始部署 example02。

在此罗列一下进入 executeDeploy(...) 的参数：stub 即 ChaincodeStub；chainname 值为 testchain；depSpec 即 CDS，但是此刻仍是被 Marshal 过的数据；policy 为 args 的第 3 个参数，值为 "OR ('Org1MSP.member','Org2MSP.member')"；escc/vscc 由于命令行未指定，为空值。

在 executeDeploy(...) 中，做了如下事情：

1）先检查了 example02 的名字、版本，是否可通过 ACL（账户控制列表）。

2）调用 lscc.getCCInstance(...)，查看 example02 是否已经存在于链上。

3）调用 ccpack, err := ccprovider.GetChaincodeFromFS(...) 将 example02 的源码读进一个

仅供非商业用途或交流学习使用

CDSPackage 对象 ccpack，对应图中的 CDSPackage。然后 cd := ccpack.GetChaincodeData()，再由 ccpack 生成一个 ChaincodeData 数据对象。

4）调用 lscc.getInstantiationPolicy(chainname, ccpack)，lscc.checkInstantiationPolicy(...)分别获取并检查一个部署策略。

5）调用 lscc.createChaincode(stub, cd)，进而直接调用 lscc.putChaincodeData(stub, cd)，传入 ChaincodeStub 和 ChaincodeData，执行部署任务。

在 putChaincodeData(stub, cd) 中，简单的检查之后，就调用 ChaincodeStub 的函数 stub.PutState(cd.Name, cdbytes)，以 example02 的名字为 key，整理过的 ChaincodeData 数据为 value，把这一对 key-value 放到账本中去。同时，从这里可以看出，链，其实就是 peer 中的账本。

stub.PutState(cd.Name, cdbytes)（core/chaincode/shim/chaincode.go 中定义），调用了 stub.handler.handlePutState(...) 触发了 lscc 的 ShimHandler 的 handlePutState() 函数。

在此罗列一下传入 handlePutState() 函数的参数：key 即 example02 的名字；value 即 ChaincodeData；txid 即交易 ID。

handlePutState() 中，所做的事情：

1）proto.Marshal(&pb.PutStateInfo{...})，首先将 key 和 value 封装，作为一个 ChaincodeMessage_PUT_STATE 类型的 ChaincodeMessage 消息的 Payload，Txid 依旧是 txid。

2）handler.sendReceive(msg, respChan)，调用 ShimHandler 将 ChaincodeMessage_PUT_STATE 类型的消息异步发送给 lscc 的 ServerHandler，然后进入 select-case 等待 ServerHandler 的回信。注意，这里等待的 respChain 是 createChannel 生成的，这个函数类似于 ServerHandler 的 createTxContext，都是以 txid 为 key 在 map 中存储通知通道，防止交易重复，且随用随删。

lscc 的 ServerHandler 收到 ChaincodeMessage_PUT_STATE 类型的消息，将只触发状态机的 enterBusyState 事件函数。该事件函数整个都是异步执行的。

最终将 key 和 value，连同处理 example02 的 lscc:1.0.0 一同写入到写集中。这里说的写集，追踪一下就可以知道，其实只是个 map，这个 map 的线路是：在 core/ledger/kvledger/txmgmt/rwsetutil/rwset_builder.go 中的 RWSetBuilder 中的 rwMap 映射，这个 map 以 lscc:1.0.0 为键，映射一个 nsRWs，而 SetState(...) 最终就是将 key 和 value 存储在这个 nsRWs 中的 writeMap（写集）中。也就是说，对 example02 的部署目前并没有真正提交到账本（数据库）中，部署也是一个交易，自然也需要最终提交到账本中，只是目前还没到最终提交的时候。至此，example02 的 chaincode 完成了真正的部署。

7. 创建镜像和启动容器

以下分析 Fabuic 创建 example02 的镜像和启动容器的过程。首先概述一下：

1）使用的是第三方库 github.com/fsouza/go-dockerclient，读者可以自行对该库进行学习，这是能理解这一步内容的基础。

仅供非商业用途或交流学习使用

2）对 core/chaincode/platforms/util/utils.go 中的 DockerBuild 注释，创建 ACC 的 docker 容器并不是简单地使用标准的 docker build+Dockerfile 的机制（因为这样产生的镜像有体积过大，有额外安全漏洞，运行笨拙等缺点），而是先积攒关于 example02 的镜像数据（Dockerfile 文件，peer 节点的 tls 证书，编译后的可执行程序），然后创建一个相对轻量级的 ACC 容器（由此可以看出，对 ACC 的容器进行减负，主要是减去要为编译 ACC 而存在的部分，这部分通常使用较少，但占用的空间和资源又相对多）。

3）编译 example02 源码用到一个容器，这个容器将 core.yaml 中 chaincode.builder 项指定的 fabric-ccenv 作为启动镜像，该镜像由 fabric 项目提供，在 Getting Started 中下载镜像时会下载（这个容器有 1G+，所以说上述的机制还是有必要的，不能为了最多 M 级别的源码一时的编译而一直运行一个 G 级别的容器），其实是一个能编译 example02 的 linux 系统容器，ccenv，就是 chaincode environment 的缩写。粗略的过程就是先创建这个容器，把 example02 的源码上传到容器中，启动容器时执行 go build…，然后下载编译好的可执行程序。

4）core/chaincode/platforms 是平台相关的代码，用于生成支持的语言的 ACC 的镜像所需的数据包，platforms.go 是总控文件，car、golang、java 是平台相关的代码，这里只关注 golang 语言。

详述过程如下：

1）**builder 执行的是 core/chaincode/platforms/platforms.go 中的 GenerateDockerBuild(...)，在这个函数中，先把 example02 容器通过 tls 连接 peer 节点的证书 peer.crt 放入 inputFiles 中，然后调用 generateDockerfile 来生成可用的 Dockerfile 文件（使用 golang 平台的 GenerateDockerfile 来创建了文件头，FROM 命令指定 example02 最终使用 **fabric-baseos 镜像，由 core.yaml 中的 chaincode.golang.runtime 项指定，ADD 将编译生成的 example02 的可执行程序压缩包 binpackage.tar 复制并解压到 /usr/local/bin 目录下，还定义了一些 LABEL 和环境变量等，也将它放入了 inputFiles 中。

2）input, output := io.Pipe() 生成了一个通道，连同 go func(){...} 中的 gw、tw 压缩对象，形成了 input<—>output<—gw<—tw 的数据流向通道，即向 tw 中写数据，最终会形成压缩包并流向 input。

3）在新启的 goroutine 中，generateDockerBuild(...) 汇总了 example02 镜像数据。先把证书和 Dockerfile 文件写入 tw，然后调用 golang 的 GenerateDockerBuild(cds, tw)，进而调用 core/chaincode/platforms/util/utils.go 中的 DockerBuild，依据 fabric-ccenv 镜像创建了一个容器，创建该容器的选项 DockerBuildOptions 指定了 3 个值：Cmd 指定了编译命令，将 example02 编译成名为 chaincode 的可执行程序并放入 /chaincode/output；InputStream 指定了输入流，该流为 example02 的 CDS.CodePackage；OutputStream 指定了容器的输出流，该流最后也通过调用 cutil.WriteBytesToPackage 写入 tw，随后流向 input。在 DockerBuild 中，所做的就是根据选项先检查 fabric-ccenv 是否存在，若不存在则尝试下载，然后创建、启动

仅供非商业用途或交流学习使用

fabric-ccenv 容器，然后等待编译完成，最后将编译好的 chaincode 从 /chaincode/output/ 中下载到输出流 OutputStream 并删除 fabric-ccenv 容器。

4）异步执行第 3 步后直接将 input 返回。返回到 builder 执行完毕将 input 返回给 reader，reader 就是接收 example 镜像数据。然后通过调用 vm.deployImage，把 reader 作为镜像的输入流（即上下文，可以理解为以此镜像使用 Dockerfile 运行容器时 Dockerfile 能使用的哪个范围下的数据），部署 example02 的镜像。这里需说明的是，这里启动的容器是 example02 镜像的 Dockerfile 指定的 fabric-baseos，且 ADD 命令会将 binpackage.tar（即名为 chaincode 的 example02 的可执行程序的压缩包）复制并解压到 /usr/local/bin 目录下，再者 createContainer 创建该容器的时候，配置 Config 中 Cmd 的值（相当于 Dockerfile 中的 CMD）是最初在 core/chaincode/chaincode_support.go 中 getArgsAndEnv(...) 生成的 args = []string{"chaincode", fmt.Sprintf("...}，所以当 client.StartContainer(containerID, nil) 启动 fabric-baseos 时（准确地说是 exmaple02 镜像，fabric-baseos 只是其基础镜像）会执行 example02 的程序 chaincode -peer.address=0.0.0.0:7051。这里要清晰地区分，执行 client.StartContainer(containerID, nil) 的是 peer 节点（这个节点可以宿存在主机中，也可以宿存在一个 docker 容器中），执行 chaincode -peer.address=0.0.0.0:7051 的是 example02 容器。

然后，创建 example02 的两个 Handler。在新运行的 example02 容器中执行 example02 的程序 chaincode，参看源码 examples/chaincode/go/chaincode_example02/chaincode_example02.go，执行的 func main 中直接调用了 shim.Start(new(SimpleChaincode))，该函数在 core/chaincode/shim/chaincode.go 中定义，相当于部署 SCC 时调用的 StartInProc，旨在启动一个 example02 的 ShimHandler 并通过 grpc 主动发送一个 ChaincodeMessage_REGISTER 类型消息给 peer 节点中 example02 的 ServerHandler。传入的 SimpleChaincode 即为 example02 链码对象，相当于 lscc 的 LifeCycleSysCC。

具体的过程如下：

1）SetupChaincodeLogging()，设置 viper 在本容器内获取环境变量值的一些方法，如把前缀设置为 CORE，把 _ 替换为 .，这样 viper.GetString("chaincode.id.name") 就可以获取启动 example02 容器时设置的 Env 中 CORE_CHAINCODE_ID_NAME=example02:antherversion 的值，还可以获取其他的环境变量以设置日志输出级别，这些环境变量均是最初在 core/chaincode/chaincode_support.go 中的 getArgsAndEnv(...) 生成的，一路被传至 example02 的容器配置中。

2）stream, err := streamGetter(chaincodename)，获取一个 grpc 流，这个流是连接 peer 节点的 ChaincodeSupport 客户端流。其中 peer 的地址是通过 flag.StringVar(&peerAddress,"peer.address"...) 获取的，最后启动 example02 容器时执行的程序是 chaincode -peer.address=0.0.0.0:7051，即通过 flag 给定了 peer 节点的地址，在此则通过 flag 获取这个地址。这一步执行过后，由于 streamGetter 中执行了 chaincodeSupportClient.Register(...)，因此 peer 节点在 core/chaincode/chaincode_support.go 中的 gprc 服务端的 Register(...) 函数将被调用，进而调用 HandleChaincodeStream。HandleChaincodeStream 新创建了属于 example02

仅供非商业用途或交流学习使用

的 ServerHandler，并调用 handler.processStream()，启动了循环接收 ShimHandler 消息的 for 循环。

3）chatWithPeer(chaincodename, stream, cc)，罗列一下传入该函数的参数：chaincodename 值为 example02:anotherversion；stream 为连接 peer 节点的 grpc 客户端流；cc 为 example02 链码对象 SimpleChaincode 自身。如同 SCC 的部署一样，通过 chatWithPeer，先创建了属于 example02 的 ShimHandler 对象，将 stream 和 cc 赋值给 ShimHandler 相应成员，然后利用 ShimeHandler 向在 peer 节点中的 ServerHandler 发送了一条 ChaincodeMessage_REGISTER 类型消息，最后启动循环接收 ServerHandler 消息的进程。

4）第 2 步中的 ServerHandler 收到第 3 步中 ShimHandler 发送的 ChaincodeMessage_REGISTER 消息，开始了注册的过程。这里的注册指的是用第 2 步新建的属于 example02 的 ServerHandler 把 prelaunchFunc() 预 Launch 的 Handler 替换掉，直到 example02 的 ServerHandler 和 ShimHandler 均达到 ready 状态。至此，第二次进行的 Launch-Execute，Launch 部分执行完毕，开始返回。直接返回定位到 core/chaincode/exectransaction.go 的 Execute(...) 中，theChaincodeSupport.Launch(...) 执行结束。

继续执行 theChaincodeSupport.Execute(...)，罗列一下传入的参数：ctxt 中依旧包含，交易模拟工具和历史查询工具；cccid 中存放 example02 的数据和最初的部署申请数据，即 SignedProposal 和 Proposal；ccMsg，一个 ChaincodeMessage_INIT 类型消息，Payload 存放的是 example02 的 CDS.CS.Input，即命令行 -c 指定的 {"Args":["init","a", "100", "b","200"]；txid 依旧是最初申请部署时的 txid，Proposal 在之后将被赋值为 SignedProposal；executetimeout 为超时时间。直接定位到 example02 容器中运行的 ShimHandler 端 core/chaincode/shim/handler.go 的 handleInit(msg)（ShimHandler 接收到 ServerHandler 发来的 ChaincodeMessage_INIT 消息，状态机触发 beforeInit 事件函数，进而调用 handleInit(msg)）。

stub.PutState 将触发 ShimHandler 的 handler.handlePutState，这时使用的 ServerHandler 和 ShimHandler 都是 example02 的，所提交的 key 是 A 的账户名，value 是 A 的余额，最终也是将这一对 key-value 通过交易模拟工具在 core/chaincode/handler.go 的 enterBusyState 中提交到 example02:anotherversion 的写集中，然后返回 core/chaincode/shim/handler.go 的 handleInit(msg) 中。随着 handler.cc.Init(stub) 的结束，触发 defer 发送 ChaincodeMessage_COMPLETED 消息。ServerHandler 收到后通知 core/chaincode/chaincode_support.go 中的 Execute(...) 结束等待并返回，exectransaction.go 中的 Execute(...) 也随之返回。

8. 广播

在 peer/chaincode/instantiate.go 的 chaincodeDeploy 中，env, err := instantiate(cmd, cf) 返回的 Envelope 紧接着被 cf.BroadcastClient.Send(env) 发送进行广播。

peer 命令的广播客户端为在 peer/common/ordererclient.go 中的 broadcastClient，封装了一个 grpc 连接和一个 AtomicBroadcast_BroadcastClient 客户端对象（这个对象其实是

AtomicBroadcastClient 客户端的一部分，即 Broadcast 流客户端，在 protos/orderer/ab.pb.go 中定义）。从所在的文件名就可知，这是连接 orderer 服务的客户端。ordering 服务客户端的地址是由命令行中 -o 指定的，若未指定，则是在 peer/common/common.go 中的 GetOrdererEndpointOfChain，通过向 cscc 发送请求获取的（这里又是一大堆文字）。这个连接在 instantiate 命令执行之初就已经被建立。

cf.BroadcastClient.Send(env) 将 env 发送到了 orderer/server.go 的 Broadcast(...) 处接收，自此数据交给了 orderer 服务中进行处理。由于涉及 orderer 服务，这里只延续讲大概过程：即 orderer 服务将 env 数据排序后发送给 peer 节点的 gossip 服务开始散播，散播后，最终提交到网络中的每个节点的链（账本）上。

9. 部署后的状态

example02 的 ChancodeData 数据被放入交易模拟器的写集中。

example02 的镜像被创建，镜像的上下文包含 Dockerfile 文件，通过 tls 连接 peer 节点的证书，可执行程序压缩包。

example02 的容器被创建，单独运行 ShimHandler，并通过 grpc 与 peer 节点中 ServerHandler 通信。

example02 的 ServerHandler 在 peer 节点的 theChaincodeSupport.runningChaincodes.chaincodeMap 中进行了注册。

example02 的 ShimHandler 和 ServerHandler 均处于 ready 状态。

example02 指定的 {"Args":["init","a", "100", "b","200"]，即 A 账户余额 100，B 账户余额 200，这两个状态被放入交易模拟器的写集中。

example02 部署的数据通过 orderer 服务排序后散播到网络的各个有效节点中，并最终提交到各自的链（账本）上（这一点本不是 peer 部署 example02 本身做的事情，只是会促成 orderer 服务和 gossip 服务要做的事情）。

5.8　Orderer 服务

5.8.1　简介

orderer 服务是单独拥有说明文档的组件，这说明这个服务应该是相当独立的。orderer 提供的服务既涉及 chaincode，也涉及账本。orderer 所提供的服务，只做 broadcast 和 deliver。orderer 节点与各个 peer 节点通过 grpc 连接，orderer 将所有 peer 节点通过 broadcast 客户端发来的消息（Envelope 格式，比如 peer 部署后的数据）按照配置的大小依次封装到一个个 block 中，并写入 orderer 的账本中，然后供各个 peer 节点的 gossip 服务通过 deliver 客户端来消费这个账本中的数据，进行自身节点账本的同步。数据流向如下：

peer 节点 ->grpc->Server->broadcast/deliver->broadcastSupport/deliverSupport->multichain->

仅供非商业用途或交流学习使用

chainSupport->kafka/solo->chain->BlockCutter->chainSupport.CreateNextBlock->chainSupport.WriteBlock->ledger 账本 ->kafka->peer 节点

涉及的目录如下：

common/

orderer/

- common：orderer 服务公用代码，主要包含了切个块的工具 blockcutter，具体的 broadcast/deliver 处理程序，各个过滤器（根据一些标准过滤要装进 block 的消息）。
- configupdate：升级配置代码。
- localconfig：本地 orderer 配置。
- ledgger：orderer 可使用的各类账本。
- kafka：卡夫卡客户端。
- solo：solo 服务端。
- multichain：总服务端，可将其视作一个总管，orderer 直接使用此对象来控制各种服务。
- server.go：orderer grpc 服务端代码。
- main.go：orderer 初始化启动代码。

protos/orderer

sampleconfig/（配置文件）

orderer 服务的命令行使用了第三方库 kingpin（v2 版本），地址为 gopkg.in/alecthomas/kingpin.v2。这里 orderer 不再像 peer 那样使用 cobra 的原因不得而知。kingpin 本身比较简单，而且 orderer 也只使用了 kingpin 最表面的一层。只有两个子命令 start 和 version，且没有任何 flag 或参数。version 不用讲，start 是默认命令，即 docker-compose-base.yaml 中启动 orderer 节点容器时所执行的命令。当前版本中，docker-compose-base.yaml 在 examples/e2e_cli/base 下，且执行的 command 是 orderer，但是 start 是默认命令，所以执行 orderer 相当于执行了 orderer start 命令。

5.8.2 模块

（1）Server

服务自身，包括 broadcast 和 deliver。有以下 3 种类型的服务，分别用于不同情况下的部署：

1）Solo ordering service。Solo，针对开发测试环境的一种服务，特点是简单，不支持 consensus，不够高效且不可测量。

2）Kafka-based ordering service、卡夫卡，一种经典的出版 - 订阅服务。orderer 中使用了第三方库 github.com/Shopify/sarama 创建客户端，以连接 kafka。在 fabric Getting Started 操作下载的镜像中，有一个 kafka 镜像以作为服务端处理消息（还有一个 zookeeper 镜像辅

仅供非商业用途或交流学习使用

助 kafka）。在产品级别的项目部署中，这种服务是当前最好的选择，有很高的吞吐量，非常高效，但是不支持容错。理解 orderer 服务中的 kafka 客户端代码需要专门了解一些关于这个系统的概念。

3）PBFT ordering service。拜占庭容错，当前处于开发之中，文档中虽未明说，但意思明显是这种服务可以兼顾高效和容错。当前版本没有相关代码，不予讨论。这个模块是服务主体，包含多个子模块，如 cutter、kafka、过滤器等。

（2）Ledger 账本

orderer 服务必须允许客户端能够在排序过的块流中进行查询，因此 orderer 服务需要一个支持账本。有以下 3 种类型的账本，分别用于不同情况下的部署：

1）File ledger。文本账本，针对产品级别的项目部署使用，默认选择此类账本（下文也只讨论这类账本）。账本直接存储在文件系统上，通过轻量级 LevelDB 数据库以 Index 进行索引，以供客户端高效地检索指定的块。

2）RAM ledger。内存账本，可配置用于存储的内存的大小，即一切数据都存于内存中。针对测试，不容错，重启后将重置。

3）JSON ledger。JSON 账本。即用 JSON 文本存储账本数据。也是针对测试环境的，特点在于简单明了，且可以容错，重启后不会重置。

（3）Profiling

监控 orderer 服务的模块，可供通过 http（如浏览器）来监控 orderer 的情况。

5.8.3　配置

配置是 orderer 服务启动过程中所必须用到的，也直接影响了 orderer 如何服务。主要涉及 orderer.yaml 和 configtx.yaml 两个配置文件。

（1）orderer.yaml

用于 orderer 服务自身。orderer.yaml 的数据使用 main.go 中 main 函数中的 conf := config.Load() 被加载，形成了 localconfig/config.go 中的 TopLevel 对象 conf（common/configtx/tool/localconfig/config.go 中也有对应的一套 TopLevel）。过程稍显复杂，多个结构体一级一级的嵌套，但较方便的地方是取出来具体的某一项配置的值的变量与 orderer.yaml 中对应的配置 key 一样，如要取出 orderer.yaml 中 GenesisFile 项的值，则使用 conf.General.GenesisFile 即可。且 config.go 中有一个默认的 TopLevel 对象 defaults 作为默认配置，可用作参考。

（2）configtx.yaml

主要供 orderer 生成 genesisblock 作为 orderer 使用的链的第一块 block。在 main.go 中 initializeBootstrapChannel 函数中生成 genesisblock 时，genesisBlock = provisional.New(genesisconfig.Load(conf.General.GenesisProfile)).GenesisBlock()（这个函数相当于 Getting Started 使用 configtxgen 手工生成的 genesis.block，使用哪种方式生成 gensisblock 由 orderer.yaml

仅供非商业用途或交流学习使用

中的 GenesisMethod 项和启动 orderer 容器时所给的环境变量，即由 docker-compose-base.yaml 中的 ORDERER_GENERAL_GENESISMETHOD 项决定，后者优先权应该大于前者）。configtx.yaml 被 genesisconfig.Load(...) 加载，形成了 ConfigEnvelope 被写入 genesisblock 中。这个过程也是相当复杂，经历了多层结构的嵌套。

5.8.4 模块初始化

1. Server 初始化

（1）gprc 服务

orderer 所基于的 gprc 服务 AtomicBroadcast 原型在 protos/orderer/ab.proto 中定义，对应生成的默认的客户端、服务端对象在 ab.pb.go 中，其中 orderer 所操作的服务端又单独实现在 orderer/server.go 中，peer 节点使用的客户端则基本沿用默认生成的客户端。server.go 中的服务端对象实例 server 在 main.go 的 main() 中，由 server := NewServer(manager, signer) 生成，使用 ab.RegisterAtomicBroadcastServer(grpcServer.Server(), server) 进行了注册，随后 grpcServer.Start() 启动。server 只是一个空架子，不做实事，其两个服务 Broadcast、Deliver 直接对应交由两个成员 handlerImpl（orderer/common/broadcast/broadcast.go）和 deliverServer（orderer/common/deliver/deliver.go）的 Handle 处理。客户端的初始化，如 peer chaincode 和 peer channel 各自的命令工厂 ChaincodeCmdFactory/ChannelCmdFactory 中的 BroadcastClient 客户端，都会随着 peer 节点的初始化而初始化。再如 peer 节点的 gossip 服务，所使用的 Deliver 客户端（core/deliverservice/deliverclient.go 中的 DefaultABCFactory）也是默认生成的客户端，会随着 gossip 模块的初始化而初始化。

（2）multiLedger 总管

它在 orderer/multichain/manager.go 中定义，是 orderer 直接使用的“总管家”，由其支配各个子模块。multiLedger 包装 chain 集合、consenters 集合、签名工具、账本生成工具、系统 chain。在 main.go 的 main() 中，manager := initializeMultiChainManager(conf, signer) 完成了 multiLedger 的初始化，初始化完成时，所有包含的子对象也相应地被初始化，所有 orderer 中现存的 chain 启动。这里的 chain 的概念比较模糊，既可以指账本本身，也可以指包含了账本的 chainSupport，还可以指具体的处理消息的流程（如 orderer/solo 和 orderer/kafka 下各自实现的 chain 所执行的 Enqueue）。而 multiLedger 自身也“人如其名”，multi+Ledger，成了多条链的“总管家”。

（3）chainSupport

ChainSupport 接口和实现均在 orderer/multichain/chainsupport.go 中定义，如同 chaincode_support 对 chaincode 一样，属于尽一切努力对 chain 操作提供支持的对象。既包含账本本身，也包含了账本用到的各种工具对象，如分割工具 cutter、过滤工具 filter、签名工具 signer、最新配置在 chain 中的位置信息（lastConfig 的值代表当前链中最新配置所在的 block 的编号，lastConfigSeq 的值则代表当前链中最新配置消息自身的编号）等。

仅供非商业用途或交流学习使用

chainSupport 也实现了一些用于支持各个工具的小接口。这样，一些流程切换、数据流向的转变的工作都交给了 chainSupport 来做。chainSupport 也做一些杂活，比如 A 工具操作中需要调用 B 工具，则 A 包含的往往不是 B 本身，而是 chainSupport，借 chainSupport 来使用 B。

（4）consenter

它分为 solo 和 kafka 两种类型，用于序列化生产（即各个 peer 节点传送来的 Envelope）出来的消息。solo 在 orderer/solo 中实现，kafka 在 orderer/kafka 中实现，两者也都起到引导和配置的作用供 chainSupport 使用，真正干活的是 solo/kafka 封装进来的 chain（orderer/solo/consensus.go）和 chainImp（orderer/kafka/chain.go）。solo 的 chain 是 for 循环 +select-case+chain 组成的简单处理流程。kafka 的 chainImp 则是一套使用第三方库 github.com/Shopify/sarama 实现的连接 kafka 服务端进行消息的订阅出版的方案，下文将详述。这里讲句题外话，solo 和 kafka 都算不上真正的 consenter，consenter 译为同意者，与官方文档中的 consensus mechanism0s 相对应。但是无论是 solo 还是 kafka 都不具备容错性，因此也就谈不上是不是 consenter 了。这个模块之所以叫这个名字，估计是在等将来的 SBFT。

（5）chain 的启动

这里 chain 指的是处理消息的线程。在 orderer/multichain/manager.go 中，当 NewManagerImpl 初始化完毕一个 chainSupport 后，都会执行 chain.start() 启动背后的 chain 处理消息的流程。

2. ledger 初始化

这里只讨论 file 形式的账本。orderer 中使用的账本同 peer 节点中使用的一样，账本生成者 BlockStoreProvider 和账本自身 BlockStore。只不过被稍微包装了一下，BlockStore 被包装进了 fileLedger（orderer/ledger/file/imp.go 中），而 fileLedger 又是 ReadWriter 接口（orderer/ledger/ledger.go 中）的实现。ReadWriter 由 Reader 和 Writer 组成，也就是说，这可能是会拓展的部分，即将来在 orderer 中可能会实现只读账本、只写账本、读写账本的分类。fileLedger 同配置工具 configResources 一同封装到 ledgerResources（orderer/multichain/manager.go）中，供 chainSupport 使用。

与 fileLedger 相配合的是同文件中的 fileLedgerIterator 迭代器，由 fileLedger 的 Iterator(startPosition) 生成，传入一个链上（实际是账本中）查询 block 开始迭代的位置赋值给迭代器的成员 blockNumber，比如 startPosition 为 0（即 SeekPosition_Oldest），则说明从链的第 1 块 block 开始遍历迭代，再比如 startPosition 为 SeekPosition_Newest，则说明从链的现存最新的一块 block 开始遍历迭代，其余的值均算为 SeekPosition_Specified 类型的位置，即任意指定链上的一个 block 块。任何查询 fileLedger 的行为都是通过循环调用这个迭代器的 Next() 进行的，每调用一次，blockNumber 自增 1。fileLedgerIterator 迭代器有一个特征是，Next() 会在迭代到还没有写入账本 block（即当前账本最新的 block 的接下来将要有的一个 block）时阻塞等待，一直等待到该 block 被写入账本中。这个阻塞控制涉及两个 chain，impl.go 中的 closedChain 和 fileLedger 的成员 signal。初始时 closedChain 被

仅供非商业用途或交流学习使用

close(closedChain) 掉。下面详述这个阻塞控制：

当迭代器当前遍历到的 block 序列号 <= 账本上当前最新的 block 序列号时（即 ledger.Height()-1），在 Next() 中，会进入 if i.blockNumber<i.ledger.Height() 分支（根据 orderer 对账本的操作，可知 ledger.Height() 返回的是链上将要添加的下一个 block 的序列号），查询后返回。而当迭代器遍历到 ledger.Height() 时，此时 i.blockNumber == i.ledger.Height()，Next() 会进入 <-i.ledger.signal 等待。当一个新的 block 要被写入账本时，则会调用 fileLedger 的 Append(block)，在 Append(block) 中，把 block 写入账本成功之后，会调用一下 close(fl.signal)（此时 Next() 的等待会结束，再次进入 if i.blockNumber<i.ledger.Height() 分支取 block），然后紧接着 fl.signal = make(chain struct{}) 再给 signal 创建一个 chain（此时若再调用 Next()，仍会再次进入 <-i.ledger.signal 等待）。

fileLedgerIterator 迭代器的 ReadyChan() 根据当前迭代器的实际情况返回。即情况 1：当 i.blockNumber>i.ledger.Height()-1 时，说明迭代器已经遍历到要查询账本的下一块还没有写入的 block 了，则返回 signal；情况 2：相反，则直接返回 closedChain。

这个控制会在 Deliver 服务处理客户端索要 block 时用到，即在 orderer/common/deliver/deliver.go 的 Handle(...) 中，正常的会进入 if seekInfo.Behavior == ab.SeekInfo_BLOCK_UNTIL_READY 分支进行 select-case 选择，此时 case<-erroredChain: 除了关闭 Deliver 时会来消息，erroredChain 不会再来消息，只会等待 case<-cursor.ReadyChain()::若情况 1，

这里将等待 signal，直到 Append(block) 中写入新的 block 后 close(fl.signal) 一下程序才会继续前进，在下文会调用迭代器的 cursor.Next()；若情况 2，由于 closedChain 是关闭的，程序将直接继续前进。

5.8.5 建立连接

grpc 分为客户端和服务端，对于 peer 连接 orderer 来说，peer 是客户端，orderer 为服务端。

Broadcast 服务主要集中在 peer 的命令中，如 peer chaincode、peer channel 命令。在 peer/common/common.go 中，GetBroadcastClientFnc 的值为 peer/common/ordererclient.go 中的 GetBroadcastClient 函数，peer chaincode 和 peer channel 命令中使用到的命令工厂中的 broadcast 客户端 BroadcastClient（在 peer/common/common.go 中的 ChaincodeCmdFactory 中）均由此函数生成。同时，peer chaincode 和 peer channel 命令执行时，都有一个 Flag，-o，来指定所要连接的 orderer 服务的地址。当命令（第一次）执行时，peer 会建立与 orderer 的 Broadcast 服务的 grpc 连接。

Deliver 服务较复杂，对象之间相互嵌套，主要集中在 peer 的 gossip 服务中，且包裹在 core/deliverservice/blocksprovider/blocksprovider.go 中的 blocksProviderImpl 中的 client 中的 createClient，client 的类型是 core/deliverservice/client.go 中的 broadcastClient，createClient 也不是 Deliver 服务本身，而是用于生成 Deliver 服务实例的。而 core/deliverservice/deliverclient.

仅供非商业用途或交流学习使用

go 中的 deliverServiceImpl，按 chainID 存储不同的 blocksProviderImpl（也就是说每个 chain 都会有一个自己的 client），是 gossip 服务直接使用的 deliver 服务的对象。

1）在 peer/node/start.go 的 serve 函数中，service.InitGossipService(...) 初始化了 gossip 服务，将一个专门生成 deliver 服务对象的 deliver 工厂赋值给 gossip 服务的成员 deliveryFactory，这个 deliver 工厂的值是 gossip/service/gossip_service.go 中的 deliveryFactoryImpl。serve 中随后的 peer.Initialize(...)（core/peer/peer.go）调用了 createChain，在 createChain 中调用了 service.GetGossipService().InitializeChannel(...)（gossip/service/gossip_service.go），传给 InitializeChannel 的最后一个参数 ordererAddresses 就是 orderer 的地址，对 gossip 服务的进一步进行了初始化。

2）在 InitializeChannel 中，使用了上一点中 deliveryFactoryImpl 的 Service 函数，进而使用 deliverclient.NewDeliverService(config)（core/deliverservice/deliverclient.go 中实现）生成了一个 Deliver 服务实例 deliverServiceImpl，赋值给 gossip 服务中的成员 deliveryService，所给的配置 config 中 Endpoints、ConnFactory、ABCFactory 被分别赋值为传进来的 orderer 的地址、core/deliverservice.go 中的 DefaultConnectionFactory、DefaultABCFactory。

3）重回 InitializeChannel 中，无论是静态指定 leader 还是动态选举 leader，最终都会调用 deliveryService.StartDeliverForChannel(...)，即在 core/deliverservice/deliverclient.go 中，StartDeliverForChannel 会使用 newClient->NewBroadcastClient（core/deliverservice/client.go），创建一个 client，再使用 blocksprovider.NewBlocksProvider(...) 创建一个包含 client 的 blocksProviderImpl，随后 go d.blockProviders[chainID].DeliverBlocks() 启动 client 的接收线程。

4）在 NewBroadcastClient 中，createClient 就被赋值为上述 config 的 ABCFactory，而生成的 broadcastClient 就是 blocksProviderImpl 中的成员 client。也就是说，blocksProviderImpl 中的 createClient 的值是 core/deliverservice/deliverclient.go 中的 DefaultABCFactory，DefaultABCFactory 即是使用了 protos/orderer/ab.pb.go 中生成的默认的 AtomicBroadcastClient（自然包含默认的 Deliver 客户端）。启动 client 的接收线程后，在 client 接收消息时，即 core/deliverservice/client.go 中的 Recv()，会执行 try(...)，进而执行 bc.doAction(action)，在 doAction 中会查看 client 是否已经和 orderer 通过 grpc 连接上，若未连接则会执行 connect() 进行 grpc 连接。

总结一下：当 peer 的容器启动的时候，peer 的 gossip 服务随着 peer node start 命令启动，其中角色是 leader 的 peer 的 gossip 服务中使用到的 deliver，服务对象 deliverServiceImpl 也会使用默认生成的 Deliver 客户端，通过 grpc 连接 orderer，并启动循环接收的线程。至于 orderer 如何不断地给 peer 中的 leader 发送 block 消息，将在 5.8.9 节中详述。

5.8.6　Broadcast

以 peer chaincode instantiate 为例。假设名为 peerA 的节点将 example02 部署完毕后，生成了一个 Envelope 发送给 orderer（参看 peer/chaincode/instantiate.go 的 chaincodeDeploy

仅供非商业用途或交流学习使用

中的 cf.BroadcastClient.Send(env))。cf 即为 peer 的命令工厂 ChaincodeCmdFactory 的实例，cf 成员 BroadcastClient 即为使用 /protos/orderer/ab.pb.go 默认生成的 Broadcast 服务客户端 AtomicBroadcast_BroadcastClient。

cf.BroadcastClient.Send(env)，通过 grpc 将消息以 Envelope 的形式发送到 orderer 节点中。

5.8.7 Orderer

Orderer（排序）主要过程如下：

1）orderer/server.go 中的 Broadcast 收到 peer 节点发来的 Envelope 消息，直接交由成员 bh 的 Handle 处理。

2）bh 原型为 orderer/common/broadcast/broadcast.go 中的 handlerImpl，Handler 函数也在这里实现。在 Handler 函数中，会在 for 中循环接收来自 peer 节点的消息：

- msg, err := srv.Recv() 接收消息。
- Unmarshal 并检查 Envelope 消息中的一些字段（如 ChannelId），如果是 HeaderType_CONFIG_UPDATE 类型的消息，则会将消息经过 bh.sm.Process(msg)，实际是调用 orderer/configupdate/configupdate.go 中的 Process 对消息进行进一步的加工，将消息加工成一个 orderer 可以处理的交易消息（加工成创建新 chain，或对已存在的 chain 进行配置更新的交易）。
- support, ok := bh.sm.GetChain(chdr.ChannelId)，获取消息对应 ChannelId 的链的 chainSupport，然后 support.Filters().Apply(msg)，使用该 chainSupport 的过滤器过滤该消息，以查看该消息是否达标，这里没有使用过滤器一并返回的 committer 集合，这是第一次过滤。
- support.Enqueue(msg)，进而调用了 support 对应的 chain 的 cs.chain.Enqueue(env) 处理消息，这个 chain 是对应的 consenter（这里只以 kafka 为例进行介绍）的 HandleChain 生成的（在创建 chainSupport 的时候，orderer/multichain/chainsupport.go 的 newChainSupport 中，cs.chain, err = consenter.HandleChain(cs, metadata)）。

3）消息进入 orderer/kafka/chain.go 的 Enqueue(env)，在最外层的 select-case 中，如果 startChan 已经被 close 则说明 orderer 的 kafka 客户端部分（即 orderer/kafka/chain.go 的 chainImpl）已经一切准备就绪，则会进入 case<-chain.startChan: 分支，否则进入 default: 分支，什么都不做（即不处理消息）而返回。在 case<-chain.startChan: 分支中，又是一个 select-case，如果该 chain 没有停止，即 haltChan 没有被 close，则会进入 default: 分支，消息也会在此进行处理。

4）在这个 default: 分支中，主要做了两个工作：

- message := newProducerMessage(chain.channel, payload) 把消息打包成 kafka 消息类型，也是使用 sarama 预定义的 ProducerMessage 类型，Topic 定义了消息的主题，Key 和 Value，key 为分区号，value 为 Marshal 过的 peer 点发来的原始的消息。

仅供非商业用途或交流学习使用

- chain.producer.SendMessage(message)，使用 sarama 的生产者对象将 kafka 消息发送到 kafka 服务端。

5）kafka 服务端（即 kafka 容器）这个“暗盒”接收到消息并生产消息。

6）在之前 chain 启动接收 kafka 服务端消息的线程中即 orderer/kafka/chain.go 中的 processMessagesToBlocks()，kafka 服务端生产的消息从 case in, ok := <-chain.channelConsumer.Messages() 处被消费出来，得到 ConsumerMessage 类型的消费消息，并进入该分支。

7）在 case in, ok := <-chain.channelConsumer.Messages() 分支中，首先出现的一个 select-case 是一个关于 errorChan 通道的开关，errorChan 通道在 chain 初始化之初是关闭的，而后在 startThread() 中的 chain.errorChan = make(chan struct{}) 中再度生成，算是再度开启。如果 errorChan 是关闭的，则在 select-case 中会进入 case <-chain.errorChan: 分支而返回，否则进入 default: 分支，程序继续。随后进入一个 switch-case，根据解压消费消息所携带的数据的类型，进入不同分支，处理消息。正常的，数据会进入 case *ab.KafkaMessage_Regular: 分支，被 processRegular(...) 处理。这里注意两个数据：

- 传入 processRegular(...) 的消费消息的 in.Offset。该值是消息在 kafka 服务端分区中的 offset，在整个生产消费过程中是序列化排序依次分配给每个消息的，因此每个消息具有唯一的 offset，所以 in.Offset 可以代表每个具体的消费消息。在 block 的结构中，一方面，Metadata 是一个数组，用来存储与 block 块有关的基础信息，其中下标 BlockMetadataIndex_ORDERER 处存储的就是 block 中最后一条消息的 offset 值 +1（也相当于下一个 block 中第一条消息的 offset）。另一方面，block 中实际存储数据的 Data 是一个 []byte 数组，序列化过后的消息都被二进制化后依次存放在这个数组中，我们可以很容易地算出 block 中具体有多少条消息。因此通过 <offset+1 的条件，就可以准确定位 block 中的每条消息。
- 传入 processRegular(...) 的 time 计时器。

8）在 processRegular() 中，将原始数据（即 peer 节点发来的消息）从 kafka 类型消息中分解出来后，交由 blockcutter 模块的 Ordered 进行分块处理。

9）消息进入 orderer/common/blockcutter/blockcutter.go 的 Ordered(env)，先用 r.filters.Apply(msg) 进行第二次过滤，然后根据配置信息分割成 block。这里假设此时满足生成一批消息的条件，该批消息和对应的 committer 集合一同被返回。

10）重回 processRegular()，当返回了一批或二批消息，程序会进入 for I,batch := range batches 循环依次处理每批消息。在循环中：

- 计算当批消息的 offset 值（即最后一条消息的 offset 值 +1），offset := receivedOffset-int64(len(batches)-i-1)，对看上面对传入的消息的 offset 的描述就可以明白为何这么计算。
- block := support.CreateNextBlock(batch)，将当批消息打包成 block，至此消息正式成块。

- support.WriteBlock(block,committers[i],encodedLastOffsetPersisted)，将 block 对应的 committer 集合，然后将 block 写入账本。

11）这里单独描述一下传入 processRegular() 中的 time 计时器和该计时器控制的主动 Cut 操作。调用 processRegular(...) 是 chainImpl 启动的 processMessagesToBlocks，而消息来源于 kafka 服务器，倘若 kafka 服务器长时间没有消息被消费出来，要么是 kafka 服务器出了故障，要么是客户端没有新的生产消息发给 kafka 服务器。在这种情况下，blockcutter 可能已经缓存了一些之前的消息，为了不使这部分消息丢失，并及时记录到账本中（比如 kafka 处理的数据量很小或消息流很不稳定，一条消息后很长时间都没有下一条消息，会造成已缓存的数据不能及时记录到账本的情况，有丢失的风险；也有交易已经产生了好长时间，但是包裹交易的消息一直悬空在 blockcutter 缓存中无法被消费，进而无法被记录和查询），configtx.yaml 中配置的 BatchTimeout 规定了超时时间，默认是 2s。当下一条消息超过 2s 还没被消费出来，将会主动 Cut 操作将现有缓存打包成一个 block 写入账本。具体的操作是：

- timer 初始状态为 nil，当消息进入 processRegular() 中，若缓存进 blockcutter 中，则会进入 if ok && len(batches) == 0 && *timer == nil 分支，设 timer 计时器为 2s 后触发，然后返回。
- 下一条消息若在 2s 之前再次进入 processRegular()，若仍是被放入缓存，则依然会进入 if ok && len(batches) == 0 && *timer == nil，重置 timer 为 2s 计时然后返回。
- 当 timer 没有及时被重置，即超过了 2s，processMessagesToBlocks 会进入 case<-timer: 分支，调用 sendTimeToCut 向 kafka 服务器发送一条 TimeToCutMessage 消息（包裹着 chain.lastCutBlockNumber+1，即若主动 Cut，会使用的 block 的序号），接着 kafka 服务器收到，然后被消费出来（这里存在一个时间过程），这条 TimeToCutMessage 消息会再次进入 processMessagesToBlocks 的 case in, ok := <-chain.channelConsumer.Messages(): 分支，进而进入 case *ab.KafkaMessage_TimeToCut: 分支，调用 processTimeToCut(...) 处理这条消息，也就是主动 Cut。
- 在 processTimeToCut(...) 中，倘若此时 if ttcNumber == *lastCutBlockNumber+1，说明 chain.lastCutBlockNumber 的值没变，也就是没有发生新的 Cut，因为一旦有新的 Cut，chain.lastCutBlockNumber 就会增 1。没有发生新的 Cut，则说明 blockcutter 中现在缓存的数据依然是发送 TimeToCutMessage 消息时的数据，或者有新数据但是依旧没有达到 Cut 的条件。此时就会进行主动的 Cut，blockcutter 中现有的缓存被打包出来后清空，timer 计时器也会重新的置为 nil。另外，在 processRegular(...) 中，当常规的触发了 Cut 后，blockcutter 中将不存在缓存消息，之后会进入 if len(batches)>0 分支，将 timer 计时器置为 nil，在 processMessagesToBlocks 中的 for-select-case 等待再次从 kafka 中消费出消息。

12）block := support.CreateNextBlock(batch) 调用了 support 中的 ledger 进行创建 block 和写入 block，在写入 block 之前会逐一执行该 block 对应的 committer 集合。至此，orderer

端处理 peer 节点发送来的消息的流程结束。

5.8.8　Deliver

Deliver 服务较 Broadcast 服务复杂。我们知道，最终 orderer 中 ledger 账本的数据会通过 Deliver 服务推送 leader 节点，现在还存在一些问题：是 orderer 主动向 peer 节点推送数据，还是 peer 节点主动向 orderer 索要数据？要多少给多少？如果是 peer 节点向 orderer 索要数据，是在何处且索要方式是什么？定时索要固定量，还是 orderer 端一有 block 产生就索要？

既然 Deliver 服务是建立在 grpc 之上的，且 orderer 为 Deliver 的服务端，则基本上可以推断是 peer 节点向 orderer 索要 block 数据。gossip 服务的索要行为的对象是 core/deliverservice/requester.go 中的 blocksRequester，索要行为发生在 RequestBlocks(...) 函数中，该函数传入一个包裹了高度值的 LedgerInfo，若该高度值为向 orderer 索要的开始的 block 的序列号，如果高度值 >0，则调用 seekLatestFromCommitter，否则调用 seekOldest()。两个函数过程基本一致，先创建一个包装了 SeekInfo 信息（即索要的 block 的起止范围信息）的 HeaderType_CONFIG_UPDATE 类型的 Envelope 消息，然后 b.client.Send(env) 向 orderer 端的 Deliver 服务端索要 block。注意，这两个函数索要的 block 范围的起点可能不一样，但是止点都一致：math.MaxUint64，即最大极限值。这相当于向 orderer 端索要现在以及将来所产生的所有 block。

索要全部的 block 的行为在下述情况发生：

1）　在 core/deliverservice/deliverclient.go 的 newClient 中，broadcastSetup := func(... 中调用了 RequestBlocks(...)。

2）broadcastSetup 作为一个参数通过 NewBroadcastClient 赋值给了 bClient（原型是 core/deliverservice/client.go 中的 broadcastClient）的成员 onConnect。

3）随后 bClient 通过 newClient 返回，在 StartDeliverForChannel(...) 中的 blocksprovider.NewBlocksProvider(...)，bClient 赋值给了对应 chainID 的 BlocksProvider（core/deliverservice/blocksprovider/blocksprovider.go）的成员 client。

4）go d.blockProviders[chainID].DeliverBlocks()，调用了 b.client.Recv() 这个 client。

5）Recv() 在 core/deliverservice/client.go 中，调用了 bc.try(...)，传给 try(...) 参数是一个执行接收动作的函数，即使用 bc.BlocksDeliverer.Recv() 接收消息（此时 BlocksDeliverer 还没有被赋值）。

6）在 try(...) 中，在 for 循环中不断尝试执行 bc.doAction(action)，这个 action 是上一步传入的接收消息的动作。在 doAction(action) 中，会首先查看连接 conn 是否已经建立，此时没有建立，则会调用 bc.connect() 建立连接。随后 resp, err := action() 执行 action 动作。

7）在 connect() 中，使用了 conn, endpoint, err := bc.prod.NewConnection() 建立了一个连接 orderer 节点的 grpc 连接，abc, err := bc.createClient(conn).Deliver(ctx) 使用建立的 grpc 连接创建了一个 Deliver 服务客户端，随后调用了 bc.afterConnect(conn, abc, cf)。

8）在 afterConnect 中，将创建与 orderer 节点的连接和 Deliver 客户端，分别赋值给 broadcastClient 的 conn（可使之后的所有 doAction(...) 中的 connect() 不再执行）和 BlocksDeliverer，然后调用 bc.onConnect(bc)，在这里发生了索要行为，向 orderer 节点索要所有的 block。

9）随后开始返回，重新返回到 doAction 中，与 orderer 节点的连接建立并发送了索要请求后，继续开始执行动作 resp, err := action()，这个 action 即为开始接收 orderer 节点发来的 block 数据。

总结一下，以上步骤都随着 gossip 服务运行起来，调用 deliverServiceImpl 对象的 StartDeliverForChannel，所启动的 go d.blockProviders[chainID].DeliverBlocks() 中的 for 循环中持续调用的，其中建立连接和索要行为只会发生一次。

执行 bc.onConnect(bc) 后，即执行的是 core/deliverservice/requester.go 中的 RequestBlocks 后，orderer 节点在 orderer/common/deliver/deliver.go 的 Handle 中的 envelope, err := srv.Recv() 处被接收。

orderer 的 Deliver 服务端向 peer 节点的 Deliver 客户端发送 block，block 进入 doAction 中，resp. err := action() 接收到 block，继续返回至 try(...)，再返回至 Recv()，再返回 core/deliverservice/blocksprovider/blocksprovider.go 的 DeliverBlocks() 中的 msg. err := b.client.Recv()，然后进入 case *orderer.DeliverResponse_Block。分支如下：

1）gossipMsg := createGossipMsg(b.chainID, payload) 将 block 包装成 gossip 服务能处理的消息类型。

2）b.gossip.AddPayload(b.chainID, payload)，将 block 添加到 peer 节点的 gossip 服务模块的本地，目的在于添加到本地的 ledger 中。再次强调，接收 block 的是 peer 节点中的 leader，所以这里的 gossip 服务模块是 leader 的。

3）b.gossip.Gossip(gossipMsg)，leader 的 gossip 服务模块向其他 peer 节点散播 block。

向 Orderer 的 Deliver 服务端索要 block 消息的，除了 gossip 服务，还有 peer channel 的几个子命令。但 peer channel 索要的都是一定量的 block，即非持续性的，只是 Deliver 服务端最后会进入 if stopNum == block.Header.Number 分支，跳出回复 block 的 for 循环。

总结一下，peer 节点中的 leader 的 gossip 服务起来之后就建立了与 orderer 的 Deliver 服务的连接（这期间 peer 节点只会在开始的时候向 orderer 的 Deliver 服务索要一次 block），之后当 orderer 中的账本中存在 block 数据后，就开始主动的向 leader 的 Deliver 客户端发送 block 数据，这个推送行为将一直持续。leader 的 Deliver 客户端收到 block 流之后会使用 gossip 服务向自身和其他 peer 节点散播这些 block 数据。另外，peer channel 等命令也会向 Deliver 服务端请求某一段 block 数据，但该推送是非持续性的。

5.8.9 orderer 共识机制

源码解析是一个自下而上的过程，从代码的实现去推敲作者的设计意图和所要解决的

问题。理解了一个方向后再去从另一个方向去理解，感觉还是很受益的。同时，建议读一下官方文档的 Bringing up a Kafka-based Ordering Service 章节，其中指导了如何部署多个 orderer 节点的步骤。

从文档中受益的是关于 orderer 服务的源码解析中没有讲述到多个 orderer 节点的情形。即区块链中，服务于每个“区域”的一个链中 peer 节点的多个 orderer 节点共同连接一个 kafka 服务器。每个 orderer 节点的服务过程如文章之前描述的那样不变，多个 orderer 连接 kafka 服务器也没有其他更多的不同之处，只是接收的“生产点”和“消费点”多了而已。但是考虑多个 orderer 节点的情况，可以提高了读者审视区块链 orderer 服务的高度，也出现了一个此前忽略的重要的概念（可以说是精髓）：共识机制。这里的共识问题就是：kafka 输出的消息被消费至多个 orderer 节点后产生的 block 块序列最终一致。

若同一个 chain 存在多个 orderer 节点服务于众 peer 节点，即“生产点”和“消费点”多了的情况，则在共识问题上主要需要解决下列两个问题：

1）每个 orderer 节点不能保证在同一时刻启动开始服务（或者有新的 orderer 加入），因此，先启动的 orderer 节点可能会先产生 block 块，但后启动的 orderer 与前启动的 orderer 间相差的 block 怎么弥补？

2）当更新 orderer 服务的配置，每个 orderer 节点不能保证同时更新配置，因此在某一时刻每个 orderer 的配置可能不同。又因为配置会影响 Cut 操作，那么如何保证一个 chain 中的某个 orderer 配置更新后，其他的 orderer 节点得到更新但不影响共识？

这里有几个前提：

fabric 中以 kafka 为基础的 orderer 实现，从实现上看是每一条链一个分区（因为每个 orderer/kafka/chain.go 中的 chainImpl 都只有一个 sarama.PartitionConsumer 成员）。在同一条链上多个 orderer 节点共同连接一个 kafka 服务器，消费 kafka 服务器上的同一个分区（可直接当成同一个文件）中的消息。

kafka 同一分区生产存储的消息序列无论在哪一个 orderer 节点，被消费出来的序列都是一样的。这也是 kafka 服务自身实现的特性。也就是说，多个 orderer 节点从 kafka 服务器出得到的消息序列，无论各个 orderer 节点消费消息的速率的快慢，最终得到的消息序列是一致的。也就是说，无论各个 orderer 节点在生产消息的速率上的差异如何，只要将不同的消息成功写入 kafka 分区，各个 orderer 节点消费出来的消息序列依旧一致。

要达到每个 orderer 节点所产生的 block 序列一致，关键在于对 Cut 操作的控制。orderer 服务中的 Cut 操作受 configtx.yaml 中的 BatchTimeout/BatchSize 两项配置的影响。

1）由 BatchTimeout 触发的 Cut 将以 TimeToCutMessage 消息的形式发送到 kafka 服务器的分区中，当一个节点接收到 TimeToCutMessage 消息时，无论该消息是否是自身发出的，只要 TimeToCutMessage 携带的 BlockNumber 值等于自身账本的下一块 block 的序列号，则进行 Cut。这相当于利用分区消息序列的唯一性设置了一个统一 Cut 的命令，任何一个 orderer 节点无论先后，只要读到这个消息就进行 Cut，这样被 Cut 出来的 block 是一致

的，只是时间早晚罢了。

2）由 BatchSize 触发的 Cut 操作，只要每个 orderer 节点的该配置一致，则顺序消费同一个分区的消息，所产生的 block 块序列自然一致。

考虑以下两种情况。

（1）问题 A

在 orderer/kafka/chain.go 中，newChain(...) 生成的 chainImpl 实例时，成员 lastOffsetPersisted 所赋的值为 -3。startThread(...) 中在生成分区消费对象，也是 chainImpl 的成员 channelConsumer 时，chain.channelConsumer, err = setupChannelConsumerForChannel(..., chain.lastOffsetPersisted+1)，最后一个参数指定的是 channelConsumer 从分区的何处开始消费（参看 sarama 库对 ConsumePartition 函数的说明），此时 chain.lastOffsetPersisted+1 值为 -2。

sarama 库中提及了两个常量：OffsetNewest，OffsetOldest。前者指从分区的最新偏移处开始消费，后者指从分区最开始偏移处开始消费。而 OffsetOldest 的常量值就是 -2，即上面所提及的 chainImpl 的 channelConsumer，默认是从分区的最开始处开始消费的。

因此，同一个 chain 有多个 orderer 节点时，每个 orderer 节点中都有自己的一个 chainImpl 对象，但一个 chain 对应一个分区，所以每个 orderer 的 chainImpl 中的 channelConsumer 消费的都是同一个分区下的消息。无论每个 orderer 节点的 chainImpl 何时开始消费消息，其都是从分区的最开始处开始消费的，因此不会错过任何一个之前（在该 orderer 节点开始消费消息之前）其他 orderer 节点产生的消息。

另外需要注意的是，kafka 服务器自身对分区消息保留的时长或大小是可以设置的，分别由 .server.properties 文件中的 log.retention.hours 和 log.retention.bytes 指定，超过设置的时间或大小，分区中之前旧的消息会被清除。Getting Started 中下载的 kafka 镜像所启动的 kafka 服务器的配置中，log.retention.hours 值为默认的 168（一周），log.retention.bytes 值为 -1，即不开启此阈值（可以容器执行 kafka 镜像，无论是否启动，都会将配置打印出来）。因此，orderer 节点最晚加入的时间也将受此限制。

（2）问题 B

如同在分区中插入 TimeToCutMessage 消息作为统一的 Cut 命令一样，一个 orderer 通过向分区中插入 ConfigUpdate 消息（这里 ConfigUpdate 消息算作一条正常的交易消息），对各个 orderer 节点进行统一的配置升级。需要明确的是，这个 ConfigUpdate 消息其实是链中具有写权限的 peer 节点通过 Broadcast 服务发给 orderer 节点的。

ConfigUpdate 消息途径 orderer/common/broadcast/broadcast.go 的 Handle(...) 时，会进入 if chdr.Type == int32(cb.HeaderType_CONFIG_UPDATE) 分支进行 bh.sm.Process(msg) 加工处理，使这个消息在发送到分区后再被各个 orderer 或前或后消费出来后到达 orderer/kafka/chain.go 的 processRegular(...) 中，ConfigUpdate 消息进入 orderer/common/blockcutter/blockcutter.go 的 Ordered(...) 中，执行了 committer, err := r.filters.Apply(msg) 而产生了消息对应的 committer。这

个 committer 的 Commit() 就是最终用来执行配置升级的。ConfigUpdate 和对应的 committer 最终会被包裹在一批消息中返回出来，假设该批消息为 batchA，对应的 committer 集合为 committerA。在 processRegular(...) 中，block := support.CreateNextBlock(batch) 将 batchA 打包成 block，随后在执行 support.WriteBlock(...) 中被依次执行了 committerA，其中自然包括 ConfigUpdate 对应的 committer。也就是说，bathA 生成的 block 是以旧的配置信息生成的最后一块 block，之后再进行 Cut，均是新的配置下所生成的 block。对于任何一个 orderer 节点，均是这样。

5.9　channel

channel 在 Fabric 中是一个相当重要的概念，可译作频道。对于 channel 的理解，不妨想象一下电视节目的频道和“我和你不在一个频道”这句话。channel 本身存在于 orderer 节点内部，但需要通过 peer 节点使用 peer channel... 命令进行维护。一个 peer 节点要想与另一个 peer 节点发生交易，最基本的前提就是两个节点必须同时处在同一个 channel 中。block 账本与 channel 也是一对一的关系，即一个 channel 对应一个账本。channel 的基本动作如表 5-1 所示。

表 5-1　channel 的基本动作

动作	释　义
Create	在 orderer 节点内部创建一个 channel。如，peer channel create -o orderer.example.com:7050 -c mychannel -f./channel.tx
Join	加入一个 channel。如，peer channel join -b mychannel.block
update	升级 channel 的某一组织的配置。如，peer channel update -o orderer.example.com:7050 -c mychannel -f,/org1MSPanchors,tx
list	列出当前系统中已经存在的所有的由 peer 节点创建 channel
fetch	获取 channel 中的 newest、oldest 块数据或当前最新的配置数据。如，peer channel fetch config config_block.pb -o orderer.example.com:7050 -c mychannel

chaincode 分为 SCC 和 ACC，这里 channel 也分为 system channel 和 application channel。system channel 是随着 orderer 节点运行之时根据 genesis.block 创建的，而通过 peer channel ... 维护的 channel，均为 application channel。对 application channel 发起维护命令的 peer 节点必须是所提交的配置文件（channel.tx，mychannel.block，Org1MSPanchors.tx）中所配置的组织中的一员，如 peer0 执行 create，则 peer0 必须是 channel.tx 中所配置的组织中的一员；而 channel.tx 对应生成的 block 块数据，就相当于 application channel 中的 genesis.block。这里所说的属于组织的一员，其实质是指该 peer 节点要持有该组织所颁发的证书。

5.9.1　目录

涉及的目录如下：

common

config：配置数据处理

configtx：配置交易处理

orderer

peer

channel：peer channel 命令源码

5.9.2 配置文件

create/join/update 三个动作都使用到了配置文件（数据）。

（1）channel.tx

它是 application channel 的创建配置文件，供 peer channel create 使用，由 configtxgen 工具根据指定的 profile 从 configtx.yaml 中读取配置数据。这里的 profile 指的是 configtx.yaml 中 Profiles 项下定义的某一个配置项。该文件主要规定了 application channel 中包含哪些组织，最终会被用作 channel 的配置原型，通过填补，作为 channel 账本的 genesis 块。例子：configtxgen -profile TwoOrgsChannel -outputCreateChannelTx ./channel.tx -channelID mychannel，指定了要生成的 application channel 的 ID 为 mychannel，并从 configtx.yaml 的 Profiles 中选取 TwoOrgsChannel 项作为 channel 的具体配置，并把生成配置数据导入 ./channel.tx 文件中。channel.tx 中存储的是二进制格式的 Envelope，这点可以参看 common/configtx/tool/configtxgen/main.go 中的 doOutputChannelCreateTx。

（2）mychannel.block

它是 application channel 的 genesis.block，供 join 使用，由 peer channel create 动作生成。上文中说 channel.tx 只是配置原型，在创建的过程中，会根据 system channel 中的配置，对 channel.tx 中的数据进行详细的填补，如填补 orderer 的配置（毕竟 application channel 是存在于 orderer 节点中的，不能对 orderer 的“规矩”一无所知），填补 application channel 所包含的组织的详细配置。填补了这些内容，最终生成一个 block，作为 application channel 的 genesis 块被保存入账本。要想 join 一个 application channel，就需要先获取这个 channel 的 genesis 块。

（3）Org1MSPanchors.tx

它是 application channel 的某组织升级配置文件，供 peer channel update 使用，由 configtxgen 工具根据指定的组织 ID 从 configtx.yaml 中指定的 profile 项中生成。channel 的配置中，基本项目之一就是频道所包含的组织了，而 update 命令就是升级 channel 配置中的某个组织的配置。例子：configtxgen -profile TwoOrgsChannel -outputAnchorPeersUpdate ./Org1MSPanchors.tx -channelID mychannel -asOrg Org1MSP，指定从 configtx.yaml 的 Profiles 下的 TwoOrgsChannel 项中获取组织 ID 为 Org1MSP 的组织的配置数据，用于升级频道 ID 为 mychannel 的 application channel，并把获取生成的配置数据导入 ./Org1MSPanchors.tx 文

件中。同样，Org1MSPanchors.tx 存储的是二进制的 Envelope，可参看 common/configtx/tool/configtxgen/main.go 中的 doOutputAnchorPeersUpdate。

5.9.3　命令

1. create/update

Chaneel 的命令 create 和 update，从字面上讲一个是创建，一个是升级，在 Fabric 都被处理成升级操作，create 作为从无到有的升级操作，update 作为从旧到新的升级操作。在 peer/channel/create.go 和 update.go 中，如果把 create 的执行过程分为三步，那么 update 只是 create 的第 1 步而已。在 create.go 的 createCmd->create->executeCreate(cf) 中，过程可以简单的分为：

1）sendCreateChainTransaction(cf)，向 orderer 发送创建 application channel 的配置数据。

2）getGenesisBlock(cf)，多次尝试从 orderer 获取生成的 application channel 的 genesis 块。即此时 application channel 已经建立，上一步发送的配置数据经过填补已经作为第 1 块 block 被保存到频道的账本中。

3）ioutil.WriteFile(file, b, 0644)，将获取的 genesis 块写入文件中，供 peer channel join 命令使用。

在 update.go 的 update(...) 中，完成的任务与 executeCreate 中的 sendCreateChainTransaction(cf) 一致，只不过发送的配置数据是用于更新 application channel 中的某个组织的配置数据。

以下将主要以 create 的执行过程为例，辅以 update 的不同之处进行讲述。

开始第 1 步。

1）在 sendCreateChainTransaction 中，根据命令行指定的 channelTxFile，也即 channel.tx 文件路径，chCrtEnv. err = createChannelFromConfigTx(channelTxFile) 读取配置数据到 chCrtEnv 中。

2）sanityCheckAndSignConfigTx(chCrtEnv)，对看函数名，理性地检查并对 chCrtEnv 进行签名。sanity 是头脑清晰，心智健全的意思，所以这里的意思就是只要你没疯，就在将配置数据发往 orderer 之前检查一下。但同时又说明，这样的检查不是非要不可，因为 channel.tx 本身是 configtxgen 工具生成的，只要工具规规矩矩不搞鬼，那这里不检查也可以。后边的步骤中还会有多次这样的 sanity 类型的检查。签名的话就是正常的签名，哪个 peer 节点发起的 peer channel create，在 chCrtEnv 中留下签名即可。

3）broadcastClient.Send(chCrtEnv)，向 orderer 节点发送 chCrtEnv。

在 orderer/server.go 的 Broadcast(...) 中的 s.bh.Handle(srv)->orderer/common/broadcast/broadcast.go 的 Handle(...) 中的 msg. err := srv.Recv()，orderer 接收到 peer 发来的原始的“配置信”msg，接着开始从 msg 中抽取数据。因为是“配置信”，因此会进入 if chdr.Type == int32(cb.HeaderType_CONFIG_UPDATE) 分支。在此分支中，msg, err = bh.sm.Process(msg)

对原始的“配置信”msg 进行了重新整理：过滤重复的配置，若是 peer channel create 的话，过滤后的配置还会被装进一个以 system channel 的 ID 为外衣的“配置信”中，形成新的“配置信”msg。

bh.sm.Process(msg) 最终调用的是 orderer/configupdate/configupdate.go 的 Process(...)。在此函数中，support, ok := p.manager.GetChain(channelID) 即获取了所需要的资源，也验证了“配置信”所对应的频道是否存在，若存在，则说明此“配置信”是用于 update 已有频道的配置的，则进入 if ok 分支执行 p.existingChannelConfig(...)；否则，则说明此“配置信”是用于创建新的频道的，则跳过 if ok 分支去执行 p.newChannelConfig(...)。这里 p.existingChannelConfig(...) 依然算是雷同于 p.newChannelConfig(...) 中的一部分。

1）在 p.newChannelConfig(...) 中，ctxm. err := p.manager.NewChannelConfig(envConfigUpdate) 这一步比较有意思，在 orderer/multichain/manager.go 的 NewChannelConfig，相当于费了九牛二虎之力验证原配置数据，并填补了一些 system channel 的配置，又到 common 下的 config 和 configtx 遛了一大圈，生成了一个内含 application channel 新的配置的 configManager（这个过程相当复杂）。但是这个 configManager 仅仅停留在当前函数中且并没有返回供调用者继续使用，而且 envConfigUpdate 虽然是以指针的形式传进去的，但是在 NewChannelConfig 并没有改变 envConfigUpdate 中的值。而下文 newChannelConfigEnv, err := ctxm.ProposeConfigUpdate(envConfigUpdate) 又直接使用原 envConfigUpdate 来生成 application channel 的频道“配置信”。如此种种会感觉 NewChannelConfig 做了很多无用功，但这里就属于上文提及的类似于 sanity 类型的验证，因为后续步骤在真正创建 application channel 时也会经历 NewChannelConfig 和创建频道所使用的 configManager，所以这里相当于事前先创建一下，若有问题趁早发现趁早返回。

2）newChannelConfigEnv. err := ctxm.ProposeConfigUpdate(envConfigUpdate)，根据原有的配置数据，利用 configManager 的 ProposeConfigUpdate 功能，通过对比“配置信”中读集和写集，将已有且版本相同的配置项过滤掉，生成要创建的 application channel 的频道配置数据 newChannelConfigEnv。这里的读集和写集可以对看生成 channel.tx 的函数 doOutputChannelCreateTx。peer channel update 所调用的 existingChannelConfig(...) 中执行的是 support.ProposeConfigUpdate(...)，与这里的 ctxm.ProposeConfigUpdate(...) 实际是一样的，都是 configManager 的 ProposeConfigUpdate，只不过 existingChannelConfig(...) 中使用的是已存在的 application channel 的 configManager（包含在 chainSupport->ledgerResources 中）的 ProposeConfigUpdate。这个已存在的 configManager 正好也证明了在创建频道时还会创建频道所使用的 configManager。

3）newChannelEnvConfig. err := utils.CreateSignedEnvelope(...)，对新生成的频道的“配置信”进行签名，类型为 HeaderType_CONFIG。peer channel update 所调用的 existingChannelConfig(...) 同样执行了这一步，且就此返回该“配置信”。

4）p.proposeNewChannelToSystemChannel(...)，将签名过的“配置信”装到一个以 system

channel为频道ID的“配置信”，且类型变为HeaderType_ORDERER_TRANSACTION，然后返回新的“配置信”。这一点是peer channel create独有的，因为创建application channel要使用system channel的chainSupport对象。

重回orderer/configupdate/configupdate.go的Process(...)中，继续对原始的“配置信”处理之后，support. ok := bh.sm.GetChain(chdr.ChannelId)，根据频道ID获取chainSupport对象，peer channel create获取的是system channel的chainSupport，peer channel update获取的是对应application channel的chainSupport。

filterErr := support.Filters().Apply(msg)，由chainSupport获取频道的过滤器集合并对“配置信”进行Apply()过滤。只讲peer channel create，获取的是system channel的过滤器集，具体为orderer启动时，在orderer/multichain/chainsupport.go的createSystemChainFilters(...)生成。依次进行非空、大小、签名的检查后，最后调用systemChainFilter的Apply()，即orderer/multichain/systemchain.go的Apply(env)。

在systemChainFilter的Apply(env)中，一系列抽取“配置信”并检查之后：

1）scf.authorizeAndInspect(configTx)对“配置信”整理和检查。这里需要明确一点，从peer发送到orderer接收至此，“配置信”均只有配置项而没有具体的配置值，是不完整的。而直到这一步才对“配置信”进行配置值的填充。在authorizeAndInspect(...)中，我们看到了NewChannelConfig，configtx.NewManagerImpl(...)的身影，即根据system channel的配置填充“配置信”对应配置项的值和新建application channel所使用的configManager。

2）return filter.Accept, &systemChainCommitter{...}，返回Accept动作和包含了完整的“配置信”的systemChainCommitter。这里注意，这一步主要目的有两个：验证和填充“配置信”，system channel的过滤器集合Apply()后返回的执行器集合被 _ 省略，是因为后文会再次生成，在block写入账本之前调用执行器。

support.Enqueue(msg)，把“配置信”作为一条消息发送给kafka（或solo），具体是由orderer/kafka/chain.go中chainImpl的Enqueue(...)发给kafka的（又由于support是system channel的chainSupport，因此这里的chainImpl是system channel对应的对象，其成员support也是system channel的chainSupport）。经过kafka暗盒的排序处理，配置信被包裹在一条KafkaMessage_Regular消息中，在orderer/kafka/chain.go的processMessagesToBlocks()中被接收，并进入case *ab.KafkaMessage_Regular:分支，交由processRegular(...)处理。

在processRegular(...)中，将“配置信”抽取出来后：

1）batches, committers, ok := support.BlockCutter().Ordered(env)，由“配置信”生成block，由于是“配置信”，因此会单独作为一个block，且support是system channel的chainSupport，获取的执行器集合committers也是system channel的过滤器集合Apply(env)之后返回的，在这里又被重新生成。执行集合中只包含systemChainCommitter。

2）for i, batch := range batches，在循环中依次处理每批消息，将每批消息打包成block，

然后 support.WriteBlock(block, committers[i],...) 在账本中写入 block。

3）support.WriteBlock(...) 执行的是 orderer/multichain/chainsupport.go 的 WriteBlock(...)，在这个函数中，首先在 for _, committer := range committers 循环中依次执行 committer.Commit()，即将执行器集合兑现。这里因为只有一个 systemChainCommitter，因此执行的是 orderer/multichain/systemchain.go 的 Commit()->scc.filter.cc.newChain(scc.configTx)，最终执行的是 orderer/multichain/manager.go 的 multiLedger 的 newChain(...)，在这里，正式的创建了 application channel。可以看到，创建 application channel 的最终效果就是：创建了频道的账本（且写入“配置信”作为 genesis 块），并在 orderer 的 multiLedger 对象成员 chains 中添加一个 chainSupport 对象，并启动了相应的服务。而上文提及的 peer channel update 使用到的 chainSupport 对象，也就是这里创建的。

4）还是在 WriteBlock(...) 中，兑现执行器集合后，cs.ledger.Append(block) 也会把 application channel 的 genesis 块写入 system channel 的账本。执行至此，peer channel create 在 orderer 端的工作基本结束。

重新返回到 orderer/common/broadcast/broadcast.go 的 Handle(...) 中，定位到 support.Enqueue(msg) 之后，之后的 srv.Send(..._SUCCESS) 就是 orderer 节点处理完毕创建新的 application channel 的工作后，向发起 peer channel create 动作的 peer 节点返回成功的应答。这里注意一下，peer 节点接收这个应答的地方是在 broadcastClient 的 Send(...) 接口内部（参看 peer/commono/ordererclient.go 的 Send(...) 实现中的 getAck()），也即当 peer/channel/create.go 的 sendCreateChainTransaction 中执行完 broadcastClient.Send(chCrtEnv)，在 Send(...) 内部已经接收到了来自 orderer 节点的应答信息。至此，第一步结束。对于 peer channel update 动作来说，至此整个动作结束。

开始第 2 步。

重新返回 peer/channel/create.go 的 executeCreate(cf) 中：block. err = getGenesisBlock(cf)，在一定时限内（默认 5s，可由 peer channel create 命令行的 -t 指定），利用 deliver 服务每隔 200 毫秒向 orderer 节点指定的 application channel 索要一次序号为 0 的 block，即 application channel 的 genesis 块，直到成功获取或超时。这一步不展开细讲。

开始第 3 步。

b. err := proto.Marshal(block)->ioutil.WriteFile(file, b, 0644)，将第 2 步获取的 block 以 application channel ID+.block 的格式写入文件，供 peer channel join 动作使用。至此，整个 peer channel create 动作执行完毕。

2. Join

Channel 的命令 peer channel join 动作看上去很像 peer 要加入 channel，而 channel 又在 orderer 节点中，所以顺理成章地就会认为 peer 需要发送一些自己的数据到 channel，然后 channel 接收这些数据后添加到自身的对象中。其实 join 的动作完全是 peer 节点本地化

自身数据和服务，以达到和对应存在于 orderer 节点的 application channel 的数据和服务相配套的一个过程。数据主要指账本，服务主要指 gossip 等模块的服务。这里假设 peer channel create 创建的是一个 ID，是 mychannel 的 application channel，对应生成的即为 mychannel.block 文件。

在 peer/channel/join.go 的 joinCmd(cf)->join(...)->executeJoin(cf) 中，主要步骤如下 :spec, err := getJoinCCSpec() 创建了一个关于 cscc 的 ChaincodeSpec 格式的“说明书”，其中的 ChaincodeInput 作为 scc 执行的输入参数，指定了两项：动作 cscc.JoinChain、从 mychannel.block 中读取的 mychannel 的 genesis 块数据。

invocation := &pb.ChaincodeInvocationSpec{...}->prop, _, err = putils.CreateProposalFromCIS(...)->signedProp, err = putils.GetSignedProposal(...)，包装 + 签名，形成一个背书申请。

cf.EndorserClient.ProcessProposal(...)，通过背书客户端，发起背书请求。最终直接定位到 core/scc/cscc/configure.go。

在 configure.go 的 Invoke(stub) 中，将进入 case JoinChain: 分支，然后是一系列的检查验证，最后执行 join 的具体动作 joinChain（cid, block）。

在 joinChain(cid, block) 中，主要动作有：

1）peer.CreateChainFromBlock(block)（core/peer/peer.go），创建 peer 节点本地的针对 mychannel 的链（账本）和 gossip 服务。Fabric 中关于 chaincode，gossip 主题中涉及使用的账本和 gossip 服务均是在此生成的。

2）peer.InitChain(chainID)（core/peer/peer.go），初始化 peer 节点本地的链。

3）producer.SendProducerBlockEvent(block)，创建事件服务，此为监控服务，在此不详述。

重回 core/scc/cscc/configure.go 的 joinChain(cid, block) 中，开始一路返回，可以直接定位回到 peer/channel/join.go 的 executeJoin(cf) 中。在接收到背书的返回结果 proposalResp 后，整个 join 动作也宣告完成。

Chapter 6 第 6 章

区块链政务数据共享及服务

6.1 背景

国务院在 2016 年 9 月 29 日发布的《关于加快推进“互联网 + 政务服务”工作的指导意见》(以下简称《意见》) 中要求，加强对电子证照、统一身份认证、网上支付等重要系统和关键环节的安全监控，提高各平台、各系统的安全防护能力，查补安全漏洞，做好容灾备份，加大对涉及国家秘密、商业秘密、个人隐私等重要数据的保护力度。《意见》指出，要推进政务信息共享。国家发展和改革委员会牵头整合构建统一的数据共享交换平台体系，贯彻执行《政务信息资源共享管理暂行办法》，打通数据壁垒，实现各部门、各层级数据信息互联互通、充分共享，尤其要加快推进人口、法人、空间地理、社会信用等基础信息库互联互通，建设政务服务数据共享平台和统一身份认证体系。国务院各部门要加快整合面向公众服务的业务系统，梳理编制网上政务服务信息共享目录，尽快向各省 (区、市) 网上政务服务平台按需开放业务系统实时数据接口，支撑政务信息资源跨地区、跨层级、跨部门互认共享。切实抓好信息惠民试点工作，2017 年年底前，在 80 个信息惠民国家试点城市间初步实现政务服务“一号申请、一窗受理、一网通办”，形成可复制可推广的经验，逐步向全国推行。《意见》明确建立健全制度标准规范。加快清理修订不适应“互联网 + 政务服务”的法律法规和有关规定，制定完善相关管理制度和服务规范，明确政务服务数据、电子公文、电子签章等的法律效力，着力解决“服务流程合法依规、群众办事困难重重”等问题。国务院办公厅组织编制国家“互联网 + 政务服务”技术体系建设指南，明确平台架构，以及电子证照、统一身份认证、政务云、大数据应用等标准规范。《意见》还要求，加强网络和信息安全保护。按照国家信息安全等级保护制度要求，加强各级政府网站信息安全建设，健全“互

联网 + 政务服务”安全保障体系。明确政务服务各平台、各系统的安全责任，开展等级保护定级备案、等级测评等工作，建立各方协同配合的信息安全防范、监测、通报、响应和处置机制。加强对政务服务数据、统一身份认证、网上支付等重要系统和关键环节的安全监控。提高各平台、各系统的安全防护能力，查补安全漏洞，做好容灾备份。建立健全保密审查制度，加大对涉及国家秘密、商业秘密、个人隐私等重要数据的保护力度，提升信息安全支撑保障水平和风险防范能力。

6.2　现有系统面临的挑战

随着政府信息化工作的不断深化和发展，中心化系统架构以及数据存储的弊端越来越多，也给政府电子政务工作带来很多挑战。

（1）政务服务数据是否可信

基于传统中心化集中管控的系统存在数据被篡改、容易造假等风险。所以政务服务数据可信度一直不高，不能有效解决信任机制的问题。

（2）信息难以全面归集

由于政府各部门存在不同职能，数据存在不同的归口管理，采用中心化系统很难从根本上对所有信息进行归集。

（3）信息难以快速复制和检索

由于中心化系统采用集中存储的方式，海量级的数据给数据存储、复制、检索带来很多挑战。

（4）信息存在安全泄露隐患

中心化系统存在系统被侵入、数据被篡改的风险。虽然可通过各种技术手段对系统安全性进行加固、防范，但系统的安全泄露的隐患始终存在。

（5）系统稳定性难度大

中心化系统由于数据存储量大，系统并发访问量高，系统的稳定性非常重要。中心系统一旦瘫痪，整个服务将不可用。这将对政府的政务工作开展产生严峻的挑战。

6.3　业务需求

随着科技的发展，社会运转和商业运转的模式均打破了时间和地域的限制，政府急需跨区域、高效率的管理和服务能力。然而，传统的中心化信息管理系统的模式，仍然存在区域限制问题、信任问题、服务稳定性问题以及全面信息归集等问题。政务服务数据共享平台必须解决以下需求。

（1）跨部门政务服务数据的归集与信任传递

虽然目前政府各部门都建设了 IT 系统，并有效管理了个人或法人的证明信息，但目前

还不能有效解决纸质证明文件通过个人或法人在政府部门间传递时信用不连续问题。通过政务服务数据共享平台的建设，可多渠道、跨部门采集政务服务数据，解决跨部门数据的归集和信任问题。

（2）需要用信息技术提升政府管理和服务的效率

利用政务服务数据共享平台的建设，建立政务服务数据目录标准体系，规范政务服务数据的管理。通过建立政务服务数据的目录体系，可实现不同权限的部门对数据的管理权限以及查看权限不同。该目录体系的建立可提升政务服务数据业务数据的质量，提高政府的办事效率，提升政府的公信力。

（3）需要全面归集信息提升政府管理和服务的能力

由于政府各个部门职能有所差别，数据归集管理不同，所以法人与自然人的政务服务数据分散在各个部门系统中。通过该共享平台的建设，要实现这些数据的全面归集，高效协作，全面提升政府的管理水平和服务能力。

（4）需要可信的数据共享系统提高证照信息的防伪能力

政务服务数据共享平台中的数据在实现共享、共建的同时，必须能提供防伪能力，实现数据的数字签名管理，提升数据的可追溯性。

（5）需要稳定的数据共享系统提高服务的稳定性

该政务服务数据共享平台需要稳定、可靠，不仅体现在对数据的处理机制、保障机制方面，更体现在整个系统的对外服务的稳定性方面。

6.4 系统总体架构设计

6.4.1 系统架构设计

区块链共享方案架构如图 6-1 所示。

其中各个模块的说明如下。

1. 管理平台中心

负责平台的可视化运营维护，包括区块链管理、政务数据共享管理和系统管理三部分。

2. 区块链节点

负责区块链底层服务，运行智能合约处理业务逻辑，所包含的业务合约如下。

1）合约管理：进行智能合约的发布、审核、管理等功能。

2）部门管理合约：区块链上部门用户的创建、删除、生成密钥等功能。

3）字典合约：为各种服务的字典数据提供服务，相当于码表的作用。

4）目录体系管理合约：政务服务数据的目录定义、元数据定义、权限控制、加解密控制等功能。

5）政务数据共享数据库：对某一政务服务数据的录入和通用查询等数据库功能。

6）积分合约：类似伪币的合约，能够查询、交易记录、加减积分等功能。

7）主体 ID 公钥合约：存储加密用户的公钥，根据主体 ID 查询公钥等功能。

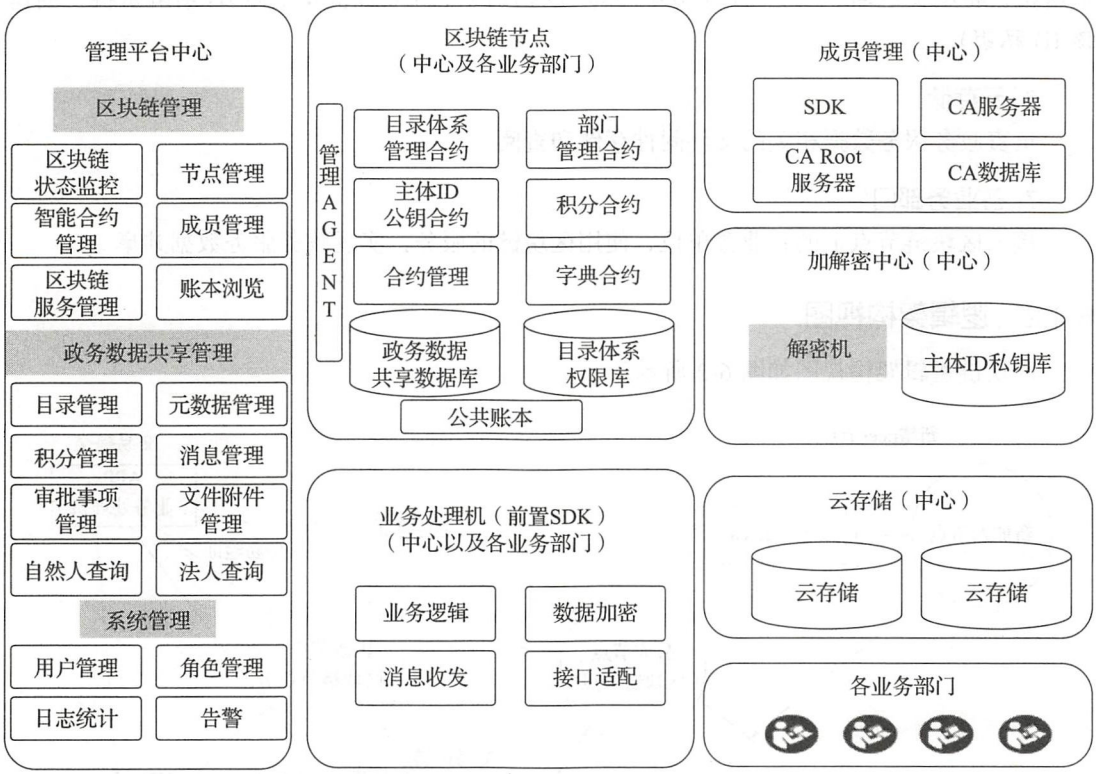


图 6-1　区块链共享方案架构

3. 业务处理机

实现主要功能如下。

1）业务逻辑：使用 GRPC 的方式建立与区块链的连接，完成用户验证、数据安全性校验、数据完整性校验，从而支持对区块链的数据交易和查询，是区块链的交易和查询前置层。

2）消息收发：使用消息接口协议，对写入消息进行 HTTP 封装，保证消息在接收者与被接收者之间仅可见。

3）接口适配：提供 Rest WebService 接口适配服务。

4）数据加密：调用加密机服务（可选）或者软件实现加解密功能。

4. 成员管理

负责区块链节点的成员管理，包括成员在线注册、成员身份验证等。

5. 加解密中心

负责证照、文件附件的加解密处理。加密机提供 API 接口，供解密服务程序调用。加

密机厂商将 API 封装成 SO 或者 DLL，进行 Linux 和 Windows 系统中的调用。共享平台会调用解密机两个 API 接口，一个是加密（解密服务给出密文和主题 ID 号，解密机查询私钥后解密，将明文返回）；另一个是分配主体 ID 私钥（业务处理机发主体 ID 给解密机，创建该 ID 私钥）。

6. 云存储

负责政务服务数据相关的文件附件存储和查阅。

7. 各业务部门

接入区块链节点上的各业务单位，使用区块链的服务，实现政务服务数据共享。

6.4.2 逻辑架构视图

区块链逻辑架构视图如图 6-2 所示。

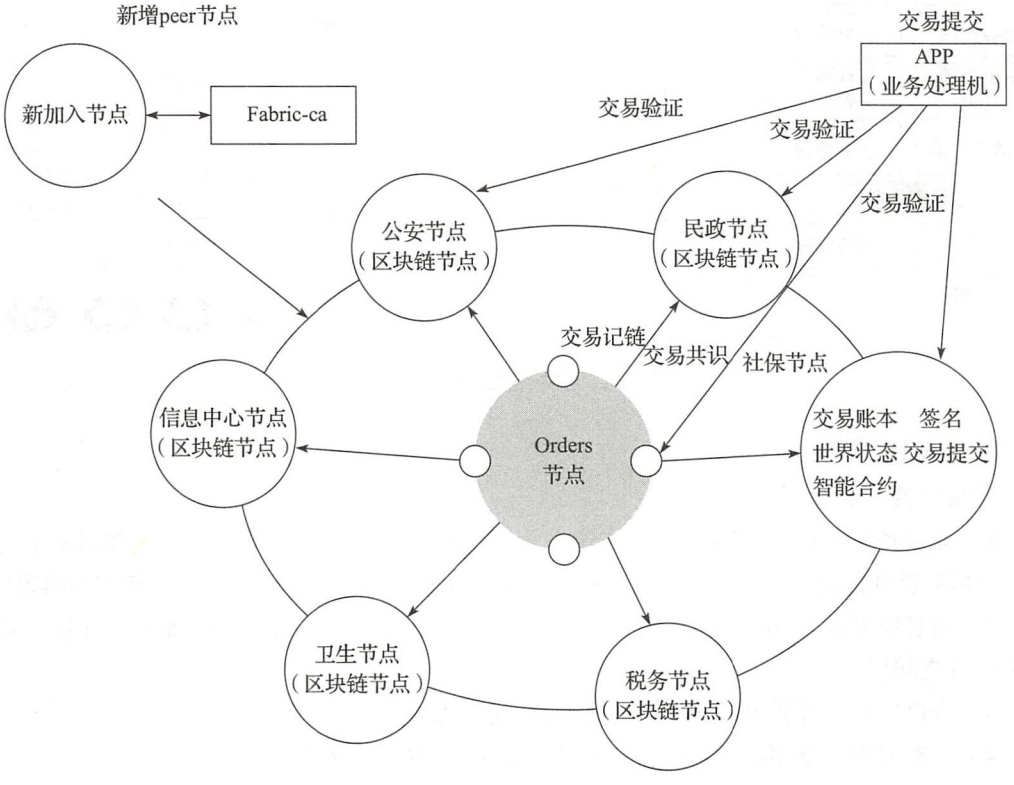


图 6-2 区块链逻辑架构视图

架构要点说明如下：

该架构为 Fabric1.0 的架构，对比原有的架构，主要提升了区块链的整体性能。具体实现上是分拆 Peer 的功能，将 Blockchain 的数据维护和共识服务进行分离，共识服务从 Peer

节点中完全分离出来，独立为 Orderer 节点提供共识服务；基于新的架构，实现多通道的结构，实现了更为灵活的业务适应性，支持更强的配置功能和策略管理功能，进一步增强系统的灵活性和适应性。

6.4.3　逻辑组网示例

一般来说，以城市为单位建设政务服务数据共享平台，为本地居民、法人提供政务服务数据办理业务，以下列出参与区块链组网的若干委办厅局节点示例，根据统筹兼顾、分步实施的原则，加入更多的委办厅局节点，组成全市统一的政务服务数据共享平台，如图 6-3 所示。

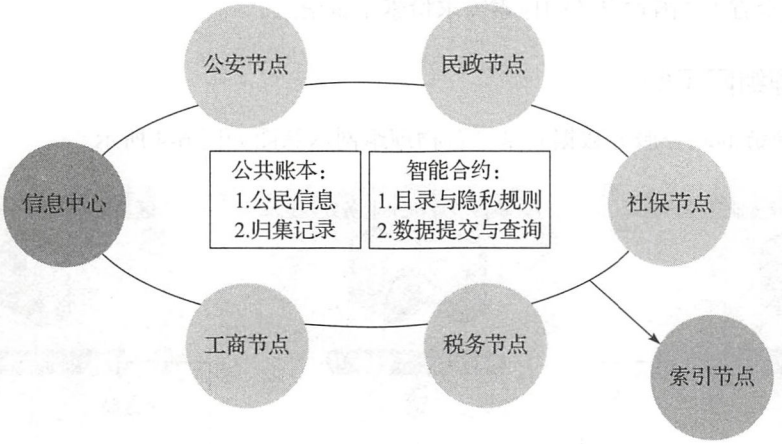


图 6-3　共享平台系统逻辑组网图

每个城市的政务服务数据共享平台建设由当地的信息中心（或信息化办）主导，信息中心的主要工作如下：

1）牵头实现政务服务数据组网。

2）牵头制定目录体系。

3）牵头制定隐私权限的智能合约。

4）分析数据、归集数据。

5）管理网络。

6）建立标准规范。

政务服务数据的组网工作如下：

1）由信息中心主导，形成城市内政务服务数据共享网络，准入的节点即可根据智能合约完善公民、法人信息，也可获得授权的证照查验服务。

2）实现全国 / 全省索引链到城市信息链的路由功能，用于从索引信息到全量详细信息的跨区域传输。

公共账本工作如下：

1）建立所有法人、公民信息的目录体系。

2）记录信息归集的过程。

3）共享全量信息到城市内所有节点。

4）抽取索引信息共享到全省 / 全国索引链。

委办厅局节点工作如下：

1）根据部门的目录体系归集内容。

2）根据授权查询证照数据。

全省 / 全国索引节点工作如下：

1）实现城市信息的索引化，并共享到全省 / 全国索引链。

2）根据全省 / 全国索引链的检索需求检索全量信息。

6.4.4 物理组网示例

基于区块链的政务服务数据共享平台物理组网示意图如图 6-4 所示。

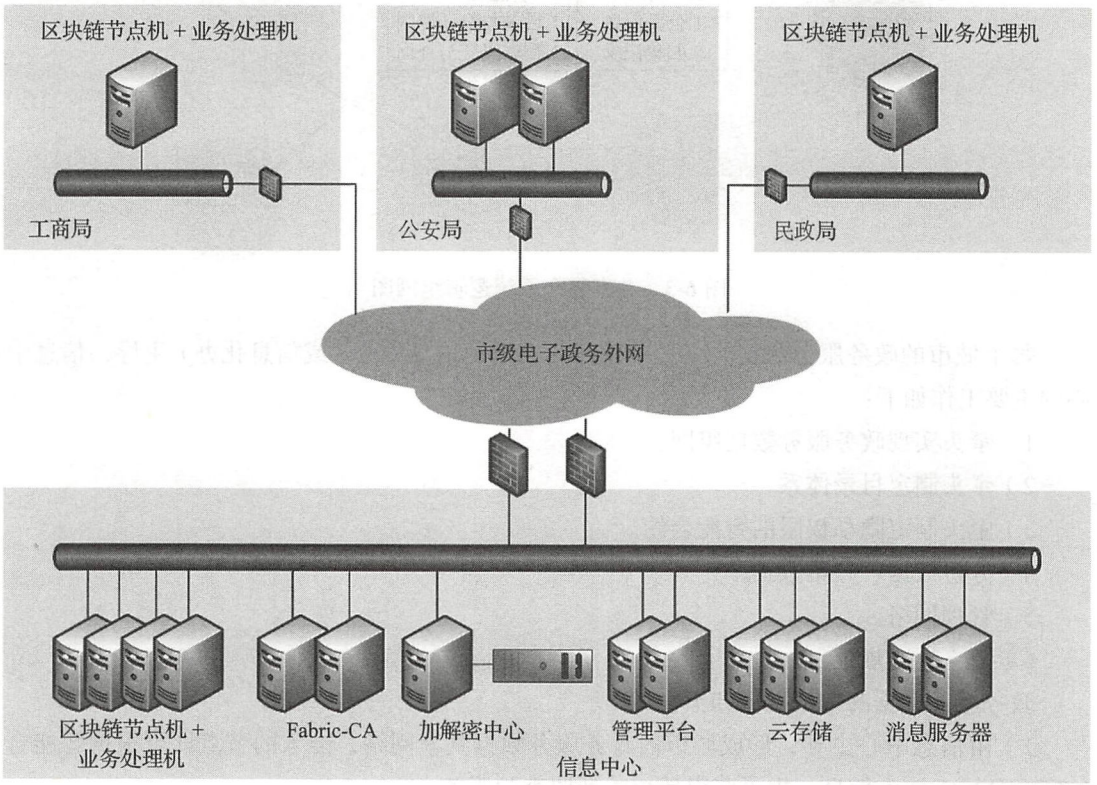


图 6-4 共享平台物理组网示意网

物理组网架构说明如下：

1）基于区块链技术的政务服务数据共享平台的网络由平台管理服务器（含数据库）、解密中心（解密服务器 + 解密机）、成员（Fabric-CA）服务器及节点服务器（节点机 + 业务处

理）等构成。在政府的各个业务部门的政务外网区域部署节点服务器。所有节点通过市级电子政务外网进行互联互通。

2）区块链节点机和业务处理机逻辑上分开，物理上可合设。

3）Fabric-CA 服务器、加解密中心、云存储、消息服务器可以是集群。如果按集群部署，需要四层交换设备作负载均衡。

4）一个部门可以是一个区块链节点，也可以由多个区块链节点分担业务。

5）多个部门可以合用一个区块链节点。

6.5　证照办件方案描述

6.5.1　场景描述

证照办件方案场景有以下两种。

1. 政务服务管理平台受理 – 部门办理 – 政务服务管理平台办结

政务服务管理平台发布业务受理的消息，相应的业务系统监听消息队列，获得订阅的消息，随后进入内部业务处理，需要的过程信息和结果信息发布到消息队列，服务管理平台根据订阅规则接收消息，如图 6-5 所示。

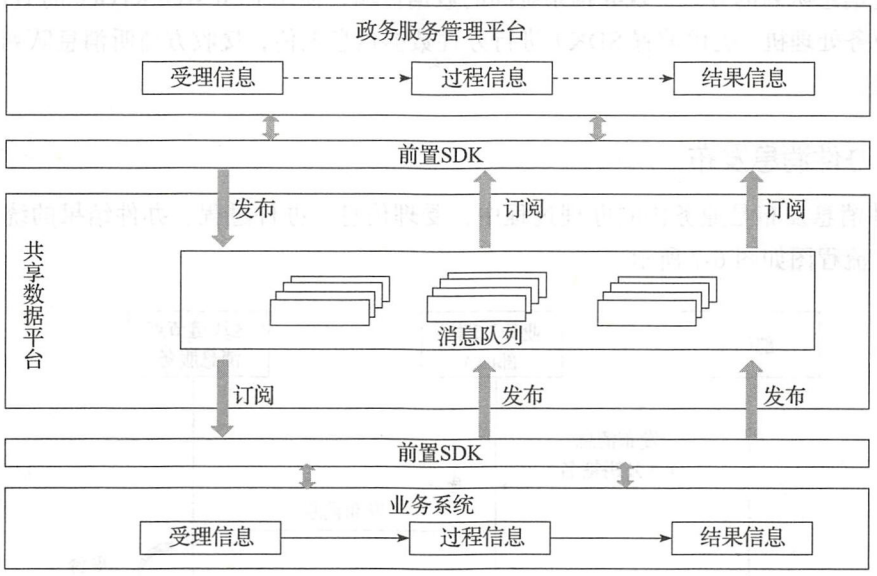


图 6-5　政务服务管理平台受理场景图

2. 部门受理 – 部门办理 – 部门办结 – 办件汇聚到政务服务管理平台

业务系统接收受理信息进入内部业务处理，同时将受理信息发布到消息队列，业务处理过程中需要提供的过程信息和结果信息发布到消息队列，服务管理平台根据订阅规则接收

消息，如图 6-6 所示。

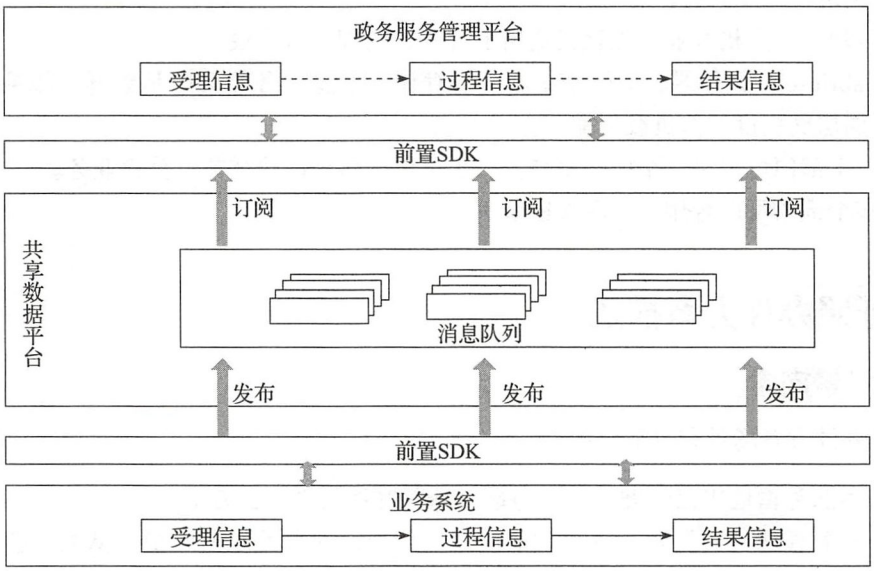


图 6-6 业务部门受理场景图

通过消息队列的方式实现异构系统间的数据传输，使用 Rest WebService 的 HTTP POST 方式与业务处理机（提供前置 SDK）进行办件数据信息上传，接收方监听消息队列并收取订阅的消息。

6.5.2 办件消息发布

办件消息发布是业务协同办理过程中，受理信息、办件过程、办件结果的统一消息发布方式，流程图如图 6-7 所示。

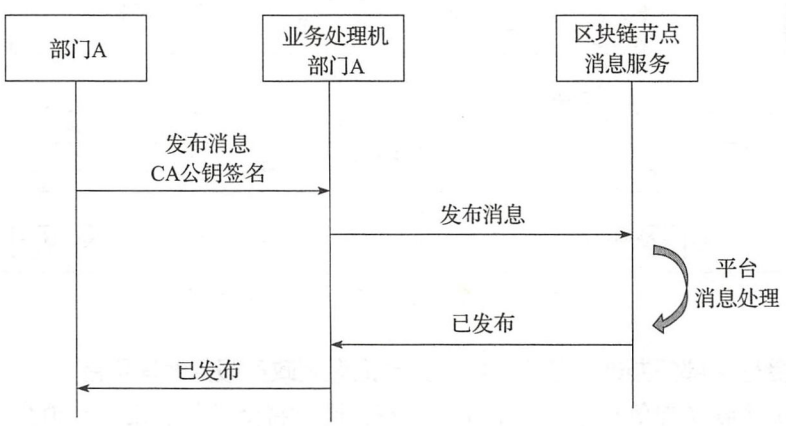


图 6-7 办件消息发送流程图

应用的场景如下：

部门发送协同办理的受理信息发布；部门订阅接收到受理信息、办件过程信息、办件结果信息，向原有发布消息的部门发送办件信息的状态。

6.5.3　可订阅消息频道查询

在本系统中，每一个部门（可细化至某个部门中的某个系统）都可以进行消息的订阅和发布，此接口用于每个接入的部门或系统可订阅的消息频道。其中，用户名、密码和订阅的消息频道应事先约定，由信息中心统一进行管理。

6.5.4　办件消息订阅

办件消息订阅流程图如图 6-8 所示。

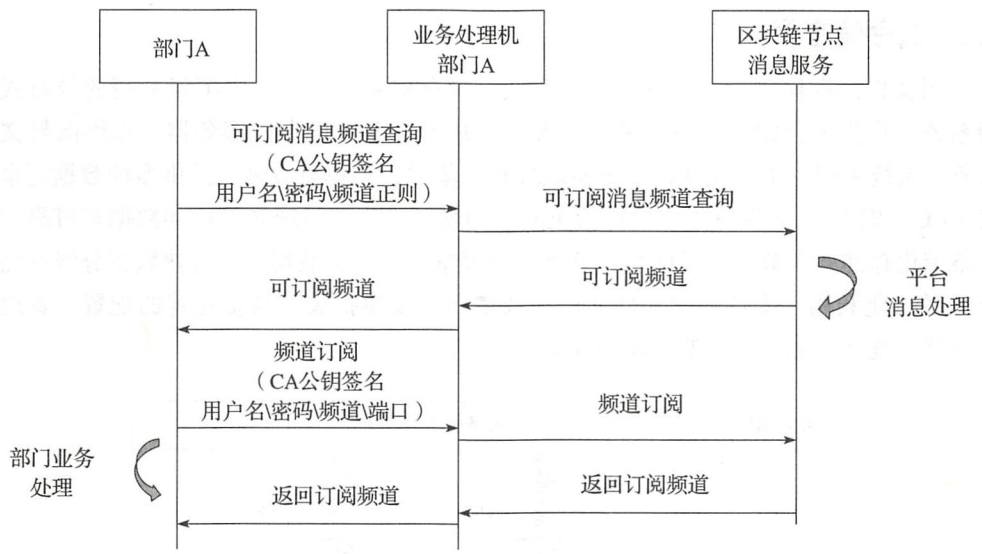


图 6-8　办件消息订阅流程图

接收消息通过消息接口协议，由各部门进行相关的开发及调试工作，需要对接进入政务内网，并申请用户名、密码和订阅的消息频道等字段，由信息中心统一进行管理，如表 6-1 所示。

表 6-1　字段表

字段	名称
IP	对接的消息服务器的 IP 地址
端口	对接的消息服务器的端口
用户名	用于订阅的用户名
密码	用于订阅的用户的密码
订阅频道	可订阅的频道可逗号分隔多个

6.6 文件共享方案

6.6.1 场景描述

完整的政务服务数据包含结构化的证照元数据和非结构的政务服务数据附件文件，比如各种证照图片、文档、视频、音频等。文件大小一般为 1 ～ 2MB，有 1GB 以上大文件（如遗嘱录像），一个城市一年的数据量在 50TB 左右，访问性能要求不高，关键需求为数据存储的高安全性，需要加密存储和防篡改。这些文件如果直接上链存储确实就能保障安全和防篡改，但是从区块链账本空间的限制、多副本的存储消耗以及业务的性能几个方面考虑，文件直接上链显然是不可取的。我们采取的是将文件的地址、Hash 上链的方案、原始的结构化文件通过文件服务存储，其地址和 Hash 存储在区块链上，在应用通过智能合约使用文件的时候，将检验文件 Hash 用于验证该文件是否被篡改。

6.6.2 云存储方案

证照文件的存储采用业界成熟的分布式文件存储系统实现。本项目采用的分布式文件存储系统，作为云存储的其中一种产品形态，基于通用 X86 存储服务器，提供海量文件存储服务。系统采用去中心化架构，全系统分布式集群、无单点故障，提供多种数据冗余备份方案（EC、副本），磁盘故障数据自动快速重构恢复，实现系统高可用和数据高可靠。数据切片条带化存储、多节点并发读写，节点之间容量和性能负载均衡，冷热数据分级存储，实现资源最大化利用。支持容量和性能的在线横向大规模扩展。具备完善的配置、管理、监控、告警、统计功能。系统架构如图 6-9 所示。

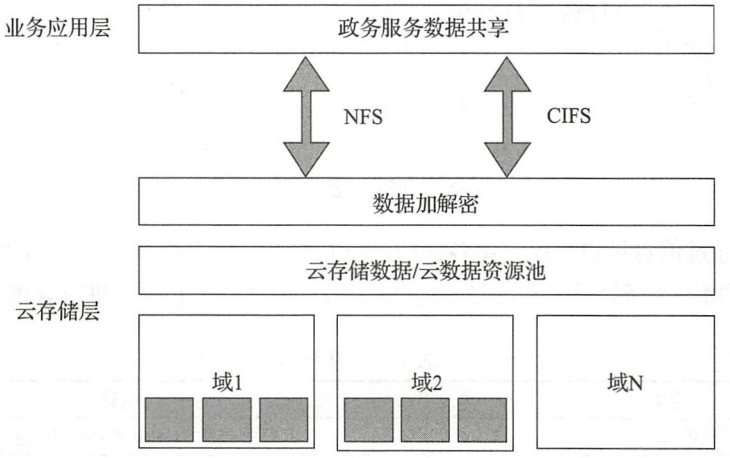


图 6-9 分布式文件系统结构图

系统架构由硬件资源层、云存储层和业务应用层三部分组成。其中硬件资源层为云存储系统部署所基于硬件资源，硬件使用统一的 X86 架构存储服务器，支持软硬件解耦；云

存储软件层为云存储系统所使用的软件资源，系统通过软件虚拟化方式，将海量的存储服务器上的物理存储空间，虚拟化成统一的用户数据存储资源池与系统元数据存储资源池，具有分布式、大容量、高性能、高可靠、弹性扩展、低 TCO 的特性。

系统通过标准的 NFS、CIFS 接口对上层政务服务数据业务系统提供云存储服务，支持任意格式文件、任意大小文件存储，支持 100PB 级存储容量、千亿文件数扩展，每 PB 存储容量提供 5GBPS 以上带宽吞吐、6 万以上 OPS 性能。同时，云存储系统提供文件加解密功能，保障政务服务数据存储的安全性，满足政务服务数据业务场景存储要求。

6.6.3　云存储安全保障方案

为保障政务服务数据在云存储中的安全性，数据进行加密存储，并将加解密功能前置部署到独立的服务器上（即加密机），如图 6-10 所示。政务服务数据业务必须通过加密机才能读写访问云存储中数据。

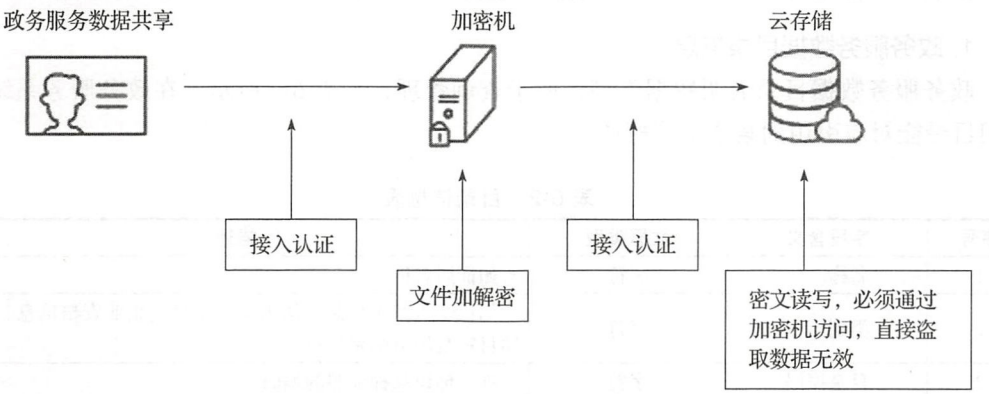


图 6-10　云存储安全保障流程图

端到端的数据安全保障方案如下：

“政务服务数据共享”访问“加密机”，进行接入认证。可选择 Ukey 证书认证，并进行细粒度的读写删查权限控制，拒绝非法用户访问。

“加密机”访问“云存储”，进行接入认证，通过公私钥对随机串加解密，并辅以人工审核，防止政务服务数据被恶意删除。

1. 加密机处理方案

上传文件时，政务服务数据业务将明文发送到加密机，加密机加密后上传云存储。下载文件时，政务服务数据业务访问加密机，加密机从云存储下载密文，本地解密后返回明文给政务服务数据业务。

加密机加密算法支持 AES 256 位密钥、SM4 国密算法等。

加密、解密过程对政务服务数据业务系统透明，加密机提供标准 NFS/CIFS 接口给政务服务数据业务系统访问，加密机与云存储之间紧密集成。

2. 存储安全

云存储采用文件分片打散存储方式，一个文件切成很多个片，均匀分布到多台服务器的多块硬盘上，并采用 EC 纠删码编码后存储，每块硬盘、每台服务器上只有每个文件的少量编码后分片，若有人盗取部分硬盘或服务器硬件，也无法恢复原始数据。

进一步，云存储端存储在硬盘上的文件和读出的文件都为加密后的密文，即使有人盗取了全部硬盘、服务器硬件，或从云存储端通过 root 用户等访问攻破，读出完整文件也无法获取文件明文内容。

6.7 证照共享方案

6.7.1 政务服务数据标准

证照共享首先要遵循政务服务数据标准进行存储和管理，然后基于统一标准进行共享。

1. 政务服务数据目录信息

政务服务数据目录表明数据类型，便于查询管理，如表 6-2 所示。在政务服务系统中，利用目录能对应引用到某个申请材料。

表 6-2 目录信息表

序号	字段含义	数据类型	描述
1	名称	字符	如证照名称
2	类别	字符	如：个人基本信息、法人基本信息、企业资格信息、投资项目审批环节结果信息
3	目录编码	字符	唯一标识某种证照的编码
4	授予对象	字符	自然人、法人、投资项目

2. 证照基本信息

证照基本信息表中记录所有证照通用的信息项，如表 6-3 所示。

表 6-3 证照基本信息表

序号	字段名	数据类型	描述
1	数据标识	字符	每个数据的唯一标识
2	目录编码	字符	
3	数据编号	字符	如证照照面上可见的唯一编号
4	颁证时间	日期	
5	有效期（起始）	日期	
6	有效期（截止）	日期	
7	颁证单位	字符	
8	持证者	字符	

（续）

序号	字段名	数据类型	描述
9	变更记录	字符	
10	数据图像	二进制	
11	电子文书	二进制	包含了完整的结构化信息和可视的证照图像

3. 证照详细信息

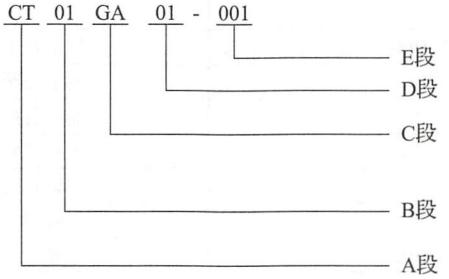
证照详细信息记录完整的信息内容，如表 6-4 所示。每种证照都由特有的信息项组成。

表 6-4　证照详细信息表

序号	字段名	数据类型	描述
1	数据标识	字符	例如，每个证照的唯一标识
2	照面字段 1	依据字段含义确定数据类型	照片字段是证照记录的个性化信息字段
3	……		
4	照面字段 n	依据字段含义确定数据类型	

4. 证照目录编码标准

证照目录编码标准如下：



其中，

A 段：地域编码，两位拼音，首字母缩写，如：XX 为某市。

B 段：目录类型编码，两位数字，01 为自然人，02 为法人。

C 段：一级目录编码，两位拼音，部门首字母缩写，如：公安为 GA。

D 段：二级目录编码，两位数字。

E 段：类型编码，3 位数字。

示例：某市房产局购房证明。

证件类型编号	目录名称	上级目录	目录类型
XX01FC01-001	购房证明	购房	自然人

常用证照编码示例如表 6-5 所示。

表 6-5 常用证照编码示例表

发证部门名称	证照名称	编码示例
某省某市公安局	户籍证明	XX01GA01-001
某省某市公安局	身份证	XX01GA01-002
某省某市公安局	护照	XX01GA01-005
某省某市公安局	港澳通行证	XX01GA01-006
某省某市公安局	台湾通行证	XX01GA01-007
某省某市公安局	驾驶执照	XX01GA03-001
某省某市公安局	机动车登记证书（个人）	XX01GA03-002
某省某市公安局	机动车行驶证（个人）	XX01GA03-003
某市国土资源局	不动产权证（个人）	XX01GT01-001
某市交通运输局	出租汽车驾驶员从业许可证	XX01JT02-001
某市民政局	结婚证	XX01MZ01-001
某市民政局	收养登记证	XX01MZ02-001

5. 元数据编码

数据基本内容元数据由数据类型编号 + 基本内容编号组成，是政务服务数据的共性属性。数据的基本内容元数据编码示例如表 6-6 所示。

表 6-6 共性属性表

内容编码	内容名称
XX01FC01-001_B00	持证者主体 ID
XX01FC01-001_B01	电子证照流水号
XX01FC01-001_B02	证照名称
XX01FC01-001_B03	证照编码
XX01FC01-001_B04	颁证时间
XX01FC01-001_B05	有效期（起始）
XX01FC01-001_B06	有效期（截止）
XX01FC01-001_B07	颁证单位
XX01FC01-001_B08	颁证者
XX01FC01-001_B09	证照变更记录
XX01FC01-001_B10	软件环境
XX01FC01-001_B11	业务行为
XX01FC01-001_B12	电子签章信息
XX01FC01-001_B13	数字证书信息
XX01FC01-001_B14	电子证照信用等级
XX01FC01-001_B15	使用状态
XX01FC01-001_B16	证照颁证机构 ID
XX01FC01-001_B17	证照颁发的区域信息，如 3201

数据扩展内容元数据由证照类型编号 + 扩展内容编号组成，是政务服务数据专属属性。数据扩展内容元数据编码示例如表 6-7 所示。

表 6-7　专属属性表

内容名称	内容编码
购房人姓名	XX01FC01-001_E01
购房人身份证号	XX01FC01-001_E02
家庭成员 1 姓名	XX01FC01-001_E03
家庭成员 1 身份证号	XX01FC01-001_E04
家庭成员 2 姓名	XX01FC01-001_E05
家庭成员 2 身份证号	XX01FC01-001_E06
家庭成员 3 姓名	XX01FC01-001_E07
家庭成员 3 身份证号	XX01FC01-001_E08
家庭成员 4 姓名	XX01FC01-001_E09
家庭成员 4 身份证号	XX01FC01-001_E10
家庭成员 5 姓名	XX01FC01-001_E11
家庭成员 5 身份证号	XX01FC01-001_E12
家庭成员 6 姓名	XX01FC01-001_E13
家庭成员 6 身份证号	XX01FC01-001_E14
该证书是否已被使用	XX01FC01-001_E15
该证书使用描述	XX01FC01-001_E16
该证书可购商品房套数	XX01FC01-001_E17
该证书符合正则法规名称	XX01FC01-001_E18

6.7.2　数据上传

部门申请 CA 证书，并通过管理平台成功注册为区块链成员后，就可以向共享平台上传政务服务数据。信息上传具体有以下三种背景：

1）各业务部门的业务系统中的大量存量数据信息按照规范上传至数据共享平台。

2）在办件过程中，用户提交的申报材料中通过部门核验的资料可上传至数据共享平台。

3）成功办件后，生成新的政务数据，例如新的电子证照等，自动上传至数据共享平台。

数据上传流程如图 6-11 所示。

数据上传前，如果有附件，业务部门会先上传附件，得到附件存储路径和 Hash。

使用 Rest WebService 的 HTTP POST 方式与业务处理机进行证照信息上传，上传信息格式为 JSON 格式，其形式与办件上传接口一致。上传后，会根据统一目录信息对信息进行编目。

6.7.3　数据查询

部门根据业务需求开发查询功能，以共享数据目录体系里的主体 ID 或元数据为查询条件，通过业务处理机传递并反馈查询条件与结果。

数据查询流程如图 6-12 所示。

业务部门通过 HTTP 方式访问附件，政务信息共享平台需要对服务请求进行验签和鉴权，确保申请方身份的合法性。

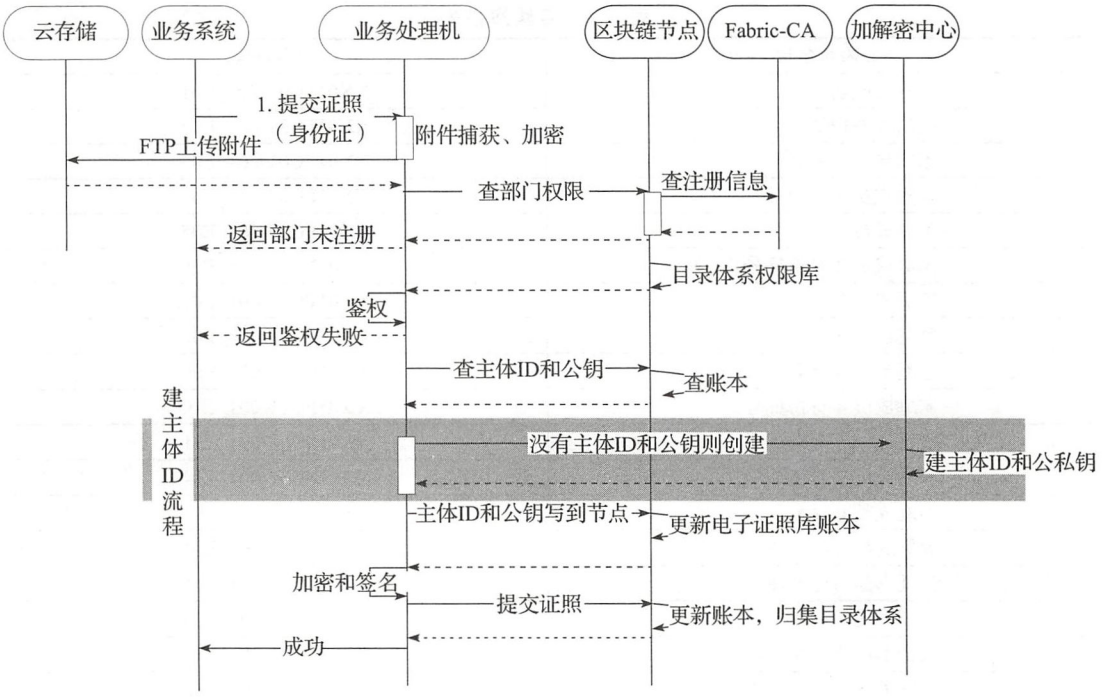


图 6-11　数据上传流程

平台支持多种方式的证照查询，包括：

（1）基于主体 ID 的查询

根据主体 ID（MAIN_ID）可查询到主体包含的政务服务数据信息。对于自然人 MAIN_ID 是身份证号，对于法人是企业 ID。

（2）基于主体和政务服务数据分类的精确查询

根据主体 ID（MAIN_ID）和政务服务数据分类编码（TYPE_NO）可获得详细的政务服务数据信息，包含政务服务数据基本信息和政务服务数据过程信息。

（3）基于数据分类的查询

根据政务服务数据分类编码（TYPE_NO）可以跨主体查询该编码下的政务服务数据信息。

（4）政务服务数据变更历史查询

根据主体 ID（MAIN_ID）和证照分类编码（TYPE_NO），可查询政务服务数据变更的历史记录。

（5）基于归属时间（证照的颁证时间）的查询

接入单位可以根据传入的时间值（ATIME）和证照分类编码（TYPE_NO），查询当日 ATIME 时间该证照分类编码（TYPE_NO）下的所有主体的该证照信息。

（6）基于归属区域的查询

接入单位可以根据传入的时间值（ATIME）和证照分类编码（TYPE_NO）及所属区域

（AREA_NO），查询当日 ATIME 时间该数据分类编码（TYPE_NO）下某个区域的政务服务数据信息。

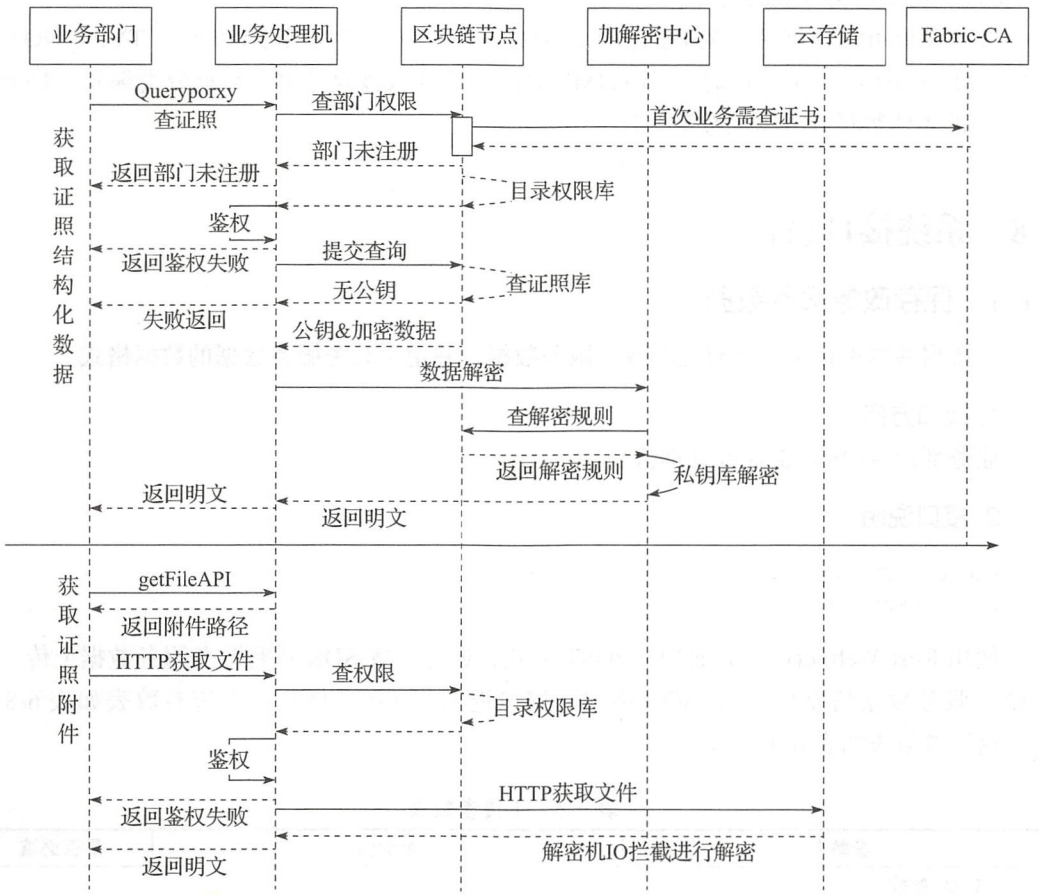


图 6-12　数据查询流程

（7）基于自定义索引的查询

可自定义索引进行查询，如在户籍中将户号 (XX01GA01-001_E11) 作为索引。

（8）基于过滤条件的政务服务数据过程记录信息查询

对于政务服务数据过程数据可以自定义过滤条件，过滤条件的个数小于等于 3 个，接入单位传入政务服务数据分类编码（TYPE_NO）和过滤条件，可以进行查询，返回多条记录。

（9）基于主体 ID 的统计

可直接查询当前系统中存在的主体数量。传入政务服务数据分类编码（TYPE_NO）可查看此目录下的主体数量和各事项数量。

（10）基于归属时间的统计

接入单位可以根据传入的时间值（ATIME）和政务服务数据分类编码（TYPE_NO），查

询当日 ATIME 时间该政务服务数据分类编码（TYPE_NO）下的主体数量和各事项数量统计。

（11）基于归属区域的统计

接入单位可以根据传入的时间值（ATIME）和政务服务数据分类编码（TYPE_NO）及所属区域（AREA_NO），查询当日 ATIME 时间、该区域该政务服务数据分类编码（TYPE_NO）下的主体数量和各事项数量统计。

6.8 系统接口设计

6.8.1 保存政务服务数据

在数据共享平台上建立对应的政务服务数据，并定义政务服务数据的数据格式。

1. 接口方向

业务部门 -> 区块链业务处理机

2. 接口说明

```
RequestURL: http://***:**/blockchain/save.htm
RequestMethod: POST
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行政务服务数据上传，上传政务服务数据信息格式为 JSON 格式，接口返回为 JSON 格式。上传参数表如表 6-8 所示，返回参数表如表 6-9 所示。

表 6-8 上传参数表

参数名	参数值	是否必填
URL 参数		
sign	对请求参数的签名字符串	
请求体参数		
HEADER	请求头部	是
HEADER.TYPE_NO	证照类型	是
HEADER.MAIN_ID	主体 ID	是
HEADER.PUBLICKEY	签名使用的私钥对应的公钥	是
BODY	证照信息	是
……		
FILES	证照附件	否
FILES.FILE_NAME	附件名称	否
FILES.FILE_PATH	附件路径	否
FILES.FILE_TYPE	附件文件类型	否
FILES .FILE_SHA256CODE	附件文件 SHA-256 码	否

表 6-9　返回参数表

参数名	参数值	值
status	服务调用是否成功	1 成功，0 失败
message	服务调用状态描述	服务调用返回描述

6.8.2　批量保存政务服务数据

在数据共享平台上建立对应的政务服务数据，并定义政务服务数据的数据格式。

1. 接口方向

业务部门 -> 区块链业务处理机

2. 接口说明

```
RequestURL: http://***:**/blockchain/save.htm
RequestMethod: POST
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行批量政务服务数据上传，上传批量政务服务数据信息格式为 JSON 格式，接口返回为 JSON 格式。上传参数表如表 6-10 所示，返回参数表如表 6-11 所示。

表 6-10　上传参数表

参数名	参数值	是否必填
URL 参数		
sign	对请求参数的签名字符串	
请求体参数		
HEADER	请求头部	是
HEADER.TYPE_NO	证照类型	是
HEADER.PUBLICKEY	签名使用的私钥对应的公钥	是
ELS	政务服务数据数组	是
ELS[].BODY	证照信息	是
……		
ELS[].FILES	证照附件	否
ELS[].FILES.FILE_NAME	附件名称	否
ELS[].FILES.FILE_PATH	附件路径	否
ELS[].FILES.FILE_TYPE	附件文件类型	否
ELS[].FILES.FILE_SHA256CODE	附件文件 SHA256 码	否

表 6-11　返回参数表

参数名	参数值	值
status	服务调用是否成功	1 成功，0 失败
message	服务调用状态描述	服务调用返回描述

6.8.3 查询政务服务数据

该接口是一个通用查询接口，可以通过请求不同的方法获取政务服务数据的相关内容。

1. 接口方向

业务部门 -> 区块链业务处理机

2. 接口说明

```
RequestURL: http://***:**/blockchain/queryporxy.htm
RequestMethod: POST
Content-Type:  application/x-www-form-urlencoded
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行政务服务数据的相关查询，返回为 JSON 格式。请求查询参数表如表 6-12 所示，返回参数表如表 6-13 所示。

表 6-12 请求参数表

参数名	参数值	是否必填	是否签名字段
DEPTNO	部门名称	是	是
FUNCTION	调用方法	是	否
publicKey	签名使用的私钥对应的公钥	是	否
sign	对请求参数的签名字符串	是	否
BASENO	数据编码	是	否
CPNO	页码	否	否
BASENO	数据编码	是	否
BASENOBOO	持证者主体 ID	是	否
b01	政务服务数据流水号	否	否
BASENO	数据编码	是	否
FILTERS	过滤条件	是	否
BASENO	数据编码	是	否
CPNO	页码	否	否
INDEXNO	索引字段	是	否
INDEXVAL	索引字段的值	是	否
OWNERTIME	归属时间	是	否
BASENO	数据编码	是	否
DAYS	天数	是	否
OWNERZONE	归属地点编号	是	否
BASENO	数据编码	是	否
BASENO	数据编码	是	否
MAIN_ID	政务服务数据主体 ID	是	否
B01	政务服务数据流水号	否	否
BASENO	数据编码	是	否
MAIN_ID	政务服务数据主体 ID	是	否
MAIN_ID	政务服务数据主体 ID	是	否
BASENO	数据编码	是	否
MAIN_ID	政务服务数据主体 ID	是	否

表 6-13　返回参数表

参数名	参数值	值
status	服务调用是否成功	1 成功，0 失败
message	结果描述	如果 status 为 1，则为返回 JSON 格式的数据 如果 status 为 0，则为出现错误的原因

6.8.4　发送消息

1. 接口方向

业务部门 -> 区块链业务处理机

2. 接口说明

```
RequestURL:  http://***:**/blockchain/SendMessage.htm
RequestMethod:  POST
Content-Type:  application/x-www-form-urlencoded
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 发送办件数据信息，上传信息格式为 JSON 格式。请求参数表如表 6-14 所示，返回参数表如表 6-15 所示。

表 6-14　请求参数表

参数名	参数值	是否必填
URL 参数		
Sign	签名字符串	是
消息体参数		
HEADER	请求头部	是
HEADER.AREA_NO	行政区划编码	是
HEADER.INTERNAL_NO	办件唯一编码	是
HEADER.RECIVED_DEPTID	接收部门编码	是
HEADER.BJ_STATU	办件状态编码	是
HEADER.A_TIME	办件业务发生的时间	是
HEADER.MAIN_ID	自然人、法人唯一编码	是
HEADER.PUBLICKEY	请求部门的公钥	
BODY	请求详细信息	是
……		
BODY.SJ_FILE	要件清单	是
BODY.SJ_FILE.NO	唯一标识	是
BODY.SJ_FILE.INTERNAL_NO	全省唯一办件编号	是
BODY.SJ_FILE.ATTAFKONLYNO	所属对象关联唯一标识	是
BODY.SJ_FILE.SJ_FILE_NAME	资料名称	否
BODY.SJ_FILE.TYPE_NO	证照类型编码	否
BODY.SJ_FILE.MAIN_ID	自然人、法人唯一编码	否
BODY.SJ_FILE.SM_FILE_NAME	上传 / 扫描件名称	是
BODY.SJ_FILE.FILE_PATH	文件地址 URL	否

（续）

参数名	参数值	是否必填
BODY.SJ_FILE.FILE_TYPE	文件格式	是
BODY.SJ_FILE.FILE_SHA256CODE	文件 Hash 码	是
BODY.SJ_FILE.CERTIFICATE_MDATA	证照属性元数据	否

表 6-15　返回参数表

参数名	参数值	值
ReturnInfo	系统运行情况	
ReturnInfo.Code	系统运行状况	0 出错，1 正常
ReturnInfo.Description	系统运行状况描述	当 ReturnInfo.Code 为 0 时的错误信息
BusinessInfo	业务执行情况	只有当 ReturnInfo.Code 为 1 的时候才会判断业务执行情况
BusinessInfo.Code	业务执行状况	0 出错，1 正常
BusinessInfo.Description	业务执行状况描述	当 BusinessInfo.Code 为 0 时的错误信息

3. 接口调用示例代码

（1）获取公私钥

通过 CA 提供的方法获取证书的公钥和私钥。获取公私钥之后，需要将公私钥转换成字节数组，再使用如下的方法转成字符串，供之后的签名和添加到 URL 中用于验签。

代码演示：获取公钥和私钥。

```
RSAPublicKey rsapublickey = 来自 CA 提供的方法；
RSAPrivateKey rsaprivatekey = 来自 CA 提供的方法；
```

（2）签名，组装消息体

将上一步中获取到的公钥放入请求的消息体中，以便前置 SDK 进行验签，将消息体转换成字符串，使用如下的签名方法进行签名：调用签名方法，使用私钥进行签名，获取签名字符串。

```
String sign=RSASignature.sign(paramStr,privateKey);
```

（3）调用消息发送接口

使用 Rest WebService 的 HTTP POST 方式提交数据，头部设置如下：

```
Content-Type: application/x-www-form-urlencoded
```

6.8.5　获取附件

在创建政务服务数据和消息发送过程中会产生很多附件，该接口可以通过这些信息获取对应的附件文件。

应用场景如下：

1）通过查询获取政务服务数据的附件信息，可以通过这些信息获取政务服务数据的附件文件。

仅供非商业用途或交流学习使用

2）通过订阅消息接收到消息，如果消息体中带有附件信息，可以通过消息体中的附件信息获取附件文件。

该接口是一个通用查询方法，可以根据代入参数的不同，而分别获取 MQ 附件和政务服务数据附件。

1. 获取消息附件接口

（1）接口方向

业务部门 -> 区块链业务处理机

（2）接口说明

```
RequestURL: http://***:**/blockchain/getFileAPI.htm
RequestMethod: POST
Content-Type:  application/x-www-form-urlencoded
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行政务服务数据的相关查询，返回为 JSON 格式。请求参数表如表 6-16 所示。

表 6-16　请求参数表

参数名	参数值	是否必填	是否签名字段
internalNo	附件所在办件流水	是	否
customerNo	部门编号	是	是
SHA256CODE	需要下载文件的 SHA256CODE	是	是
ATime	服务请求发生时间	是	是
sign	签名字符串	是	否
publicKey	签名证书的公钥	是	否

返回参数：

返回一个字符串，如果成功则是一个下载文件的地址，如果失败则是失败的说明。

如果返回一个下载地址则可以直接使用该链接进行文件下载，调用方法 GET。

（3）接口调用示例代码

1）获取公私钥。

```
    String pfxPath = 证书文件路径;
    String publicKey = Util.getHexString(RSAEncrypt.loadPublicKeyByFile(pfxPath,
"123456").getEncoded());
    String privateKey = Util.getHexString(RSAEncrypt.loadPrivateKeyByFile(pfxPath,
"123456").getEncoded());
```

2）签名。将 customerNo、SHA256CODE、ATime 放入 Map<String,String> 对象中，并对这个 Map 里面的所有参数进行排序后签名。

```
    Map<String,String>paramMap = new HashMap<String, String>();
    paramMap.put("customerNo", customerNo);
```

仅供非商业用途或交流学习使用

```
paramMap.put("SHA256CODE", SHA256CODE);
paramMap.put("ATime", ATime);
Map<String,String>signMap = Util.sortMapByKey(paramMap);
String content = Util.getParams(signMap);
String sign=RSASignature.sign(content,privateKey);
```

将生产的签名和证书的公钥添加到请求参数中。

```
content+= "&sign=" + sign + "&internalNo=" + internalNo + "&publicKey=" +
publicKey;
```

3）调用获取下载地址接口。使用 Rest WebService 的 HTTP POST 方式提交数据，头部设置如下：

```
Content-Type: application/x-www-form-urlencoded
```

2. 获取政务服务数据附件接口

（1）接口方向

业务部门 -> 区块链业务处理机

（2）接口说明

```
RequestURL: http://***:**/blockchain/getFileAPI.htm
RequestMethod: POST
Content-Type:  application/x-www-form-urlencoded
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行政务服务数据附件的相关查询，返回为 JSON 格式。请求参数表如表 6-17 所示。

表 6-17 请求参数表

参数名	参数值	是否必填	是否签名字段
BASENO	证照编号	是	是
B01	流水	否	否
MAIN_ID	主体 ID	是	是
SHA256CODE	需要下载文件的 SHA256CODE	是	是
ATime	服务请求发生时间	是	是
sign	签名字符串	是	否
publicKey	签名证书的公钥	是	否

返回参数：

返回一个字符串，如果成功则是一个下载文件的地址，如果失败则是失败的说明。

如果返回一个下载地址则可以直接使用该链接进行文件下载，调用方法 GET。

（3）接口调用示例代码

1）获取证书的公钥和私钥。

```
String pfxPath = "D:/CA/block_chain.pfx";
```

仅供非商业用途或交流学习使用

```
    String publicKey = Util.getHexString(RSAEncrypt.loadPublicKeyByFile(pfxPath,
"123456").getEncoded());
    String privateKey = Util.getHexString(RSAEncrypt.loadPrivateKeyByFile(pfxPath,
"123456").getEncoded());
```

2）签名。将 BASENO、MAIN_ID、SHA256CODE、ATime 放入 Map<String,String> 对象中，并对这个 Map 里面的所有参数进行排序后签名。

```
Map<String,String>paramMap = new HashMap<String, String>();
paramMap.put("BASENO", BASENO);
paramMap.put("MAIN_ID",MAIN_ID );
paramMap.put("SHA256CODE", SHA256CODE);
paramMap.put("ATime", ATime);
Map<String,String>signMap = Util.sortMapByKey(paramMap);
String content = Util.getParams(signMap);
String sign=RSASignature.sign(content,privateKey);
```

3）调用获取下载地址接口。

使用 Rest WebService 的 HTTP POST 方式提交数据，头部设置如下：

```
Content-Type: application/x-www-form-urlencoded
```

6.8.6　获取可订阅消息

在本系统中，每一个部门（可细化至某个部门中的某个系统）都可以进行消息的订阅和发布功能，此接口用于每个接入的部门或系统可订阅的消息频道。其中，用户名、密码和订阅的消息频道应事先约定，由信息中心统一进行管理。

1. 接口方向

业务部门 -> 区块链业务处理机

2. 接口说明

```
RequestURL:  http://***:**/blockchain/QueryMessage.htm
RequestMethod:  POST
Content-Type:  application/x-www-form-urlencoded
```

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行可订阅消息查询，请求参数表如表 6-18 所示，返回参数表如表 6-19 所示。

表 6-18　请求参数表

参数名	参数值	是否必填	是否签名字段
userName	MQ 链接用户名	是	是
passWord	MQ 链接密码	是	是
channels	用户可以监听的消息队	是	是
sign	对请求参数的签名字符串	是	否
publicKey	签名使用的私钥对应的公钥	是	否

仅供非商业用途或交流学习使用

表 6-19 返回参数表

参数名	参数值	值
status	服务调用是否成功	1 成功，0 失败
Description	服务调用状态描述	当 status 为 0 的时候接口返回的描述
queueNamesList	查询到的消息队列名称	队列名称数组

3. 实现接口调用

获取公钥和私钥：

```
String publicKey = Util.getHexString(rsapublickey.getEncoded());
String privateKey = Util.getHexString(rsaprivatekey.getEncoded());
```

将需要签名的参数放入一个 Map<String,String> 对象中：

```
Map<String,String>paramMap = new HashMap<String, String>();
paramMap.put("userName", USERNAME);
paramMap.put("passWord", PASSWORD);
paramMap.put("channels", CHANNEL);
```

先调用 util 中的排序方法 sortMapByKey，再调用 getParams 方法将参数取出来，最后调用 RSASignature 中的 sign 方法进行签名，获取到签名字符串。

```
Map<String,String>signMap = Util.sortMapByKey(paramMap);
String content = Util.getParams(signMap);
String sign = RSASignature.sign(content, privateKey);
```

6.9 系统功能设计

6.9.1 总体功能结构

基于区块链的政务服务数据共享平台总体功能如表 6-20 所示。

表 6-20 基于区块链的政务服务数据共享平台总体功能描述

一级功能	二级功能	功能描述
政务服务数据业务功能	目录管理	服务目录内容的编目、审核发布等
	目录授权	分目录的上传数据权限和分目录的查询数据权限
	部门类别管理	创建新部门，设定部门类别，约束部门类别权限
	元数据管理	元数据的增删改查
	保存政务服务数据	在数据共享平台上建立对应的政务服务数据
	批量保存政务服务数据	在数据共享平台上批量建立对应的政务服务数据
	数据查询	提供通用的查询接口，通过请求不同的方法获取政务服务数据的相关内容
	数据详情查看	查看某个数据的详细字段
区块链管理功能	节点管理	管理节点设备的运行状态，节点的增删改查
	智能合约管理	管理在线运行的智能合约的代码即链代码，发布新代码
	账本浏览	按区块或交易 ID 查询，浏览区块链上公共账本
	部门管理	实现接入部门的登记、维护等相关功能
	审批事项管理	支持新增审批事项，查询审批事项清单，审批、查询审批事项详情
	会员积分管理	支持会员积分查询、更改、积分变化历史记录查询功能

仅供非商业用途或交流学习使用

（续）

一级功能	二级功能	功能描述
系统管理功能	用户管理	实现系统用户的增加、删除、修改、查询
	角色管理	实现系统角色的增加、删除、修改、查询，实现角色的权限分配
	日志、告警、系统状态监控	日志、告警、系统状态监控，提供系统维护能力

6.9.2　政务服务数据业务功能

1. 目录管理

目录管理系统将国家《政务信息资源目录体系》(GB/T21063-2007）中定义的编目系统、目录管理系统进行整合，主要实现了服务目录内容的编目、审核发布等功能。

（1）目录分类

将一级政务服务信息资源目录分为自然人信息、法人信息、证照信息。也可依据实际的应用需要进行分类。

（2）编目功能

一级政府资源目录由管理部门维护，用于跨部门、跨层级部门信息共享的索引，二级部门内部目录由部门自己设定，用于部门内部信息共享的索引。编制完成之后提交审核。

（3）审核发布

对目录类别的审核、对目录项审核、对目录文字审核、对目录流程图审核、资源目录项中标识符编码的查询显示、数据资源目录项中标识符编码的人工修改。

（4）目录查询

包括多维度目录查询、列表查询、信息资源访问功能。

（5）目录维护

对已发布的目录进行维护、删除、停用、更新、重组目录等操作。

目录管理示意图如图 6-13 所示。

图 6-13　目录管理示意图

仅供非商业用途或交流学习使用

2. 目录授权

目录权限管理包括分目录的上传数据权限和分目录的查询数据权限，是为区分和授予所有数据在不同部门的权限，以满足数据在不同部门对目录数据提交、目录浏览、数据检索的权限要求。通过区块链共享平台将目录体系的权限同步到所有组网节点，节点应用功能对目录体系内的数据进行相关操作时，根据业务部门在目录体系的权限进行授权，高效、有效以实现权限控制。

目录授权示意图如图 6-14 所示。

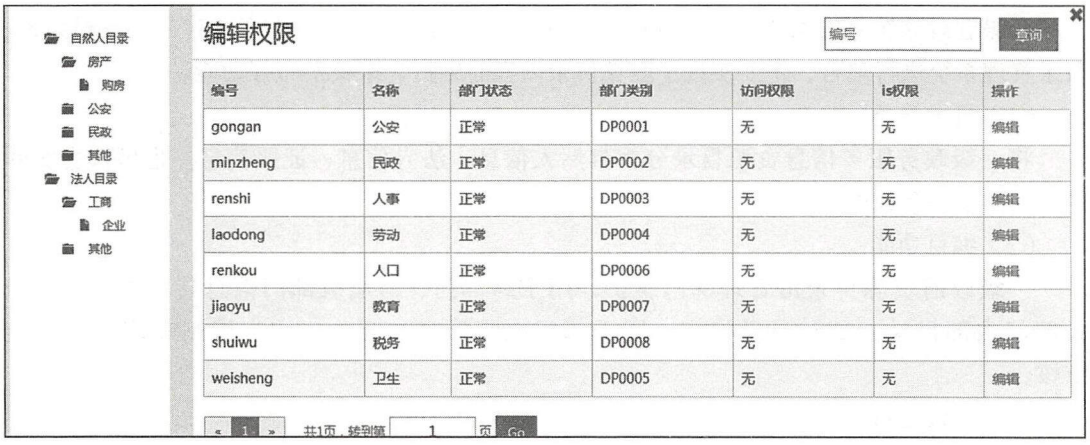


图 6-14 目录授权示意图

3. 部门分类管理

部门类别管理可以为创建新部门设定部门类别，约束部门类别权限。

部门分类管理示意图如图 6-15 所示。

图 6-15 部门分类管理示意图

仅供非商业用途或交流学习使用

4. 元数据管理

对政务信息资源的标识、内容、分发、数据质量、数据表现、数据模式、图示表达、限制和维护等信息进行统一管理，以利于发现与定位信息资源，管理与整合信息资源，改进系统有效存储、检索的能力。

发挥区块链技术数据溯源的特征，对元数据的初始上传者进行签名，记录原始上传者、过程修改者、信息查询查看者等过程，并将元数据的相关过程分布式存储在组网内，确保对元数据的所有操作都有据可查，提高系统的可追溯性和稳定性。

元数据管理包括以下功能。

（1）元数据定义

包括了基础分类信息制定、元模型制定、数据分层定义、数据主题管理、模型规范制定。

（2）元数据查询

元数据查询必须支持对元数据库中的元数据基本信息进行查询与检索的功能，可查询维表、指标、过程及参与的输入输出对象信息，以及其他纳入管理的对象基本信息，查询的信息按处理的层次及业务主题进行组织，查询功能返回实体及其所属的相关信息。

（3）元数据维护

平台的元数据是动态更新的，因此元数据的维护需提供对元数据的增加、删除和修改等基本操作。元数据维护示意图如图 6-16 所示。

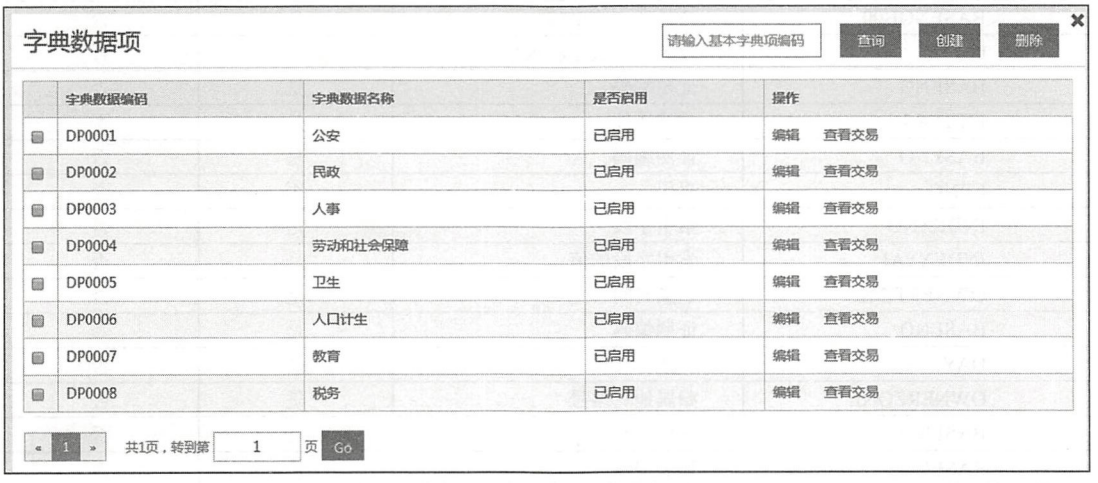


图 6-16　元数据维护示意图

5. 保存政务服务数据

在数据共享平台上建立对应的政务服务数据，并定义政务服务数据的数据格式。通过政府业务部门系统提交自然人信息，需经过业务系统、政务服务数据共享平台业务处理机、区块链节点服务器等环节，合法的业务请求将把新的数据记录到区块链数据库上。目录体系

仅供非商业用途或交流学习使用

数据规范是跨部门协作的基础，也是区块链数据存储的基础。目录体系数据规范包括自然人的基础信息以及扩展信息的规范。规范中定义了数据字段的代码、数据类型、长度、是否允许为空等。业务系统提交时自然人数据时，必须遵守规范，否则区块链节点拒绝接收。

6. 批量保存政务服务数据

在数据共享平台上批量建立对应的政务服务数据，并定义政务服务数据的数据格式。

7. 查询

提供通用的查询接口，可以通过请求不同的方法获取政务服务数据的相关内容。

使用 Rest WebService 的 HTTP POST 方式，通过前置 SDK 进行政务服务数据的相关查询，返回为 JSON 格式。请求参数表如表 6-21 所示，返回参数表如表 6-22 所示。

表 6-21 请求参数表

参数名	参数值	是否必填	是否签名字段
DEPTNO	部门名称	是	是
FUNCTION	调用方法	是	否
publicKey	签名使用的私钥对应的公钥	是	否
sign	对请求参数的签名字符串	是	否
BASENO	证照编码	是	否
CPNO	页码	否	否
BASENO	证照编码	是	否
BASENOB00	持证者主体 ID	是	否
b01	政务服务数据流水号	否	否
BASENO	证照编码	是	否
FILTERS	过滤条件	是	否
BASENO	证照编码	是	否
CPNO	页码	否	否
INDEXNO	索引字段	是	否
INDEXVAL	索引字段的值	是	否
OWNERTIME	归属时间	是	否
BASENO	证照编码	是	否
DAYS	天数	是	否
OWNERZONE	归属地点编号	是	否
BASENO	证照编码	是	否
BASENO	证照编码	是	否
MAIN_ID	政务服务数据主体 ID	是	否
B01	政务服务数据流水号	否	否
BASENO	证照编码	是	否
MAIN_ID	政务服务数据主体 ID	是	否
MAIN_ID	政务服务数据主体 ID	是	否
BASENO	证照编码	是	否
MAIN_ID	政务服务数据主体 ID	是	否

仅供非商业用途或交流学习使用

表 6-22　返回参数表

参数名	参数值	值
status	服务调用是否成功	1 成功，0 失败
message	结果描述	如果 status 为 1，则为返回 JSON 格式的数据，详见下面的示例。 如果 status 为 0，则为出现错误的原因

查看政务服务数据示意图如图 6-17 所示。

图 6-17　查看政务服务数据示意图

8. 证照详情查看

有权限的账号可以查看某个证照的详细字段，而无权限的账号则无法查看相关证照的详细信息。

查看证照详细信息示意图如图 6-18 所示。

6.9.3　平台管理功能

1. 节点管理

管理节点设备的相关功能包括：

1）对所有节点服务器有集中监控的功能，如：机器 IP 管理、运行状态管理、存储空间监控等。

2）对节点服务器的启动、禁用等操作功能。

3）对节点服务器故障警报的功能。

节点管理示意图如图 6-19 所示。

仅供非商业用途或交流学习使用

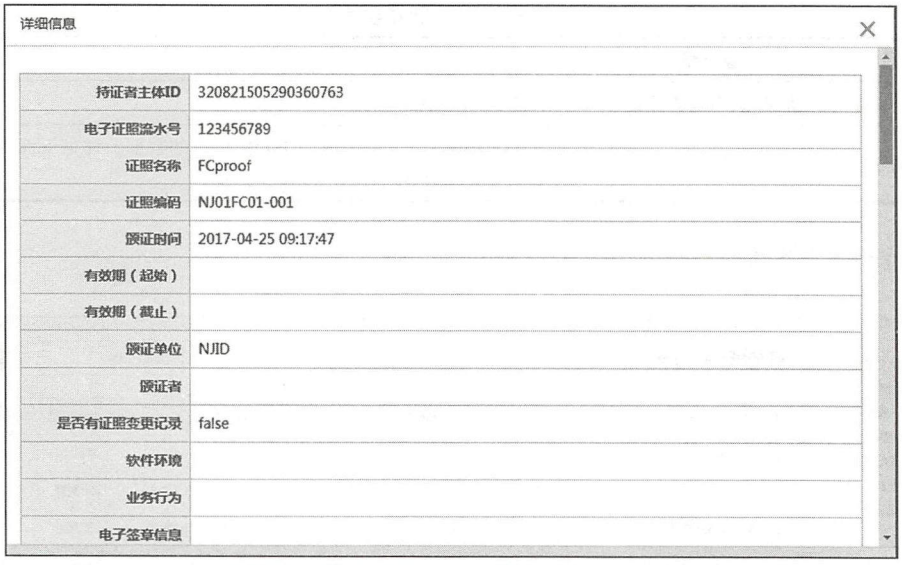


图 6-18 查看证照详细信息示意图

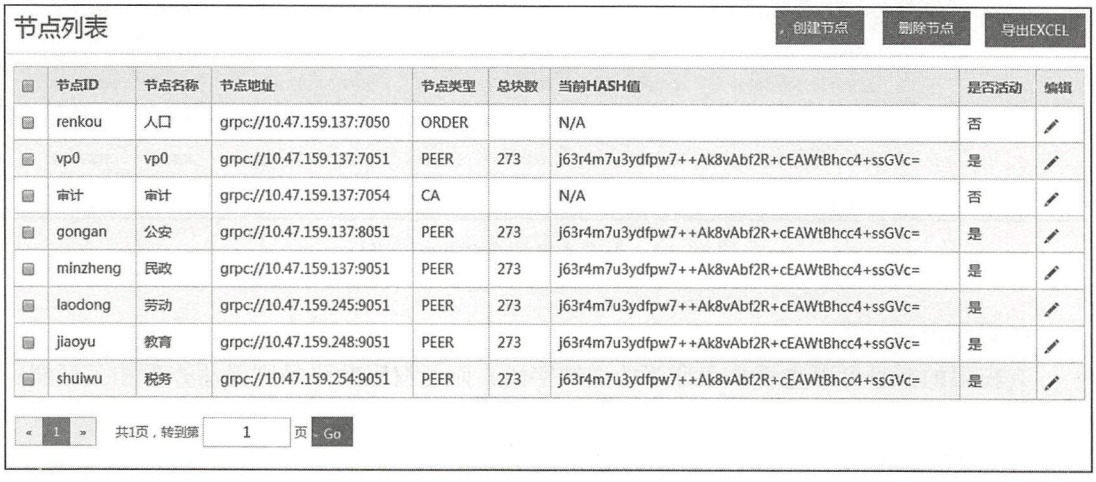


图 6-19 节点管理示意图

2. 智能合约管理

区块链网络在同步记账的基础上，所有业务功能均由动态发布的智能合约（链代码）支持，本平台在智能合约的管理方面的功能如下：

1）浏览查看目前区块链平台的所有智能合约，可以查看合约的发布人、发布时间、调用 API 文档、合约的法律文本描述、合约的源代码等信息。

2）可以通过管理功能增加并上传新的智能合约。

3）可以通过管理功能上传智能合约的版本。

智能合约管理示意图如图 6-20 所示。

仅供非商业用途或交流学习使用

智能合约管理　　新增

调用名	别名	版本	发布时间	合约描述	交易ID	操作
AccessChaincode	AccessChaincode	1.0	2017-09-13 03:28:14		f8a1567abed7903f3474...	
CryptChaincode	CryptChaincode	1.0	2017-09-13 03:28:31		97cec134e06d3b6db787...	
DictChaincode	DictChaincode	1.0	2017-09-13 03:28:46		01869b4ae92716c0b860...	
ELicenseChaincode	ELicenseChaincode	1.0	2017-09-13 03:29:03		13049a80bef145f275c4...	
ListChaincode	ListChaincode	1.0	2017-09-13 03:29:19		1e51de313995f896be14...	
PointChaincode	PointChaincode	1.0	2017-09-13 03:29:37		bcdbfafd60d33e1bd0f2...	

« 1 »　共1页，转到第 1 页 Go

图 6-20　智能合约管理示意图

3. 账本浏览

区块链通过去中心化的分布式记账，保证有全网所有节点的账本内容一致。账本浏览功能可以连接任意节点，浏览区块链账本的每个区块与交易内容。

1）按区块查看本区块的所有的交易内容。

2）通过交易 ID 定位至该交易所在的区块，并展示区块的详细内容。

3）当存在某一节点账本不一致情况时，通过浏览账本分析不一致原因。

4）追溯业务数据时，通过浏览账本的原始交易信息，以证明数据的不可篡改，通过交易的发起签名证明交易发起者的身份。

账本浏览示意图如图 6-21 所示。

首页　第260页　第261页　第262页　第263页　第264页　第265页　第266页　第267页　第268页　第269页　第270页　第271页　第272页　尾页

区块序号	区块时间	本块hash	前块hash	操作
272	2017-09-13 05:18:25	IVw0Oj/9HgIBv+d8gLytTZz07NExsQ9SGe5wB332o/k=	8HXIiI92j7IU4NdNyKHHrPrwqwHifhVnTkkn1BoUqYE=	查看

交易记录

交易时间	交易通道	交易请求方	交易合约	操作
2017-09-13 05:18:24	rbcchannel		ELicenseChaincode:1.0	查看
2017-09-13 05:18:24	rbcchannel		ELicenseChaincode:1.0	查看
2017-09-13 05:18:24	rbcchannel		ELicenseChaincode:1.0	查看
2017-09-13 05:18:24	rbcchannel		ELicenseChaincode:1.0	查看
2017-09-13 05:18:24	rbcchannel		ELicenseChaincode:1.0	查看

« 1 2 3 4 5 10 »　共10页，转到第 1 页 Go

图 6-21　账本浏览示意图

仅供非商业用途或交流学习使用

4. 部门管理

部门管理对接业务部门管理智能合约，实现接入部门的登记、维护等相关功能，包括：

1）参与方注册与审批功能，接入部门注册、审批和权限分配等相关功能。

2）接入部门的状态变更功能，接入部门状态变更的功能。

3）接入部门公私钥管理，实现对接入部门分配公私钥（CA 证书）的流程。

部门管理示意图如图 6-22 所示。

部门列表

请输入部门编号 查询 新增 导出Excel 导入会员

编号	名称	部门状态	部门类别	部门角色	操作
admin	admin	正常	未知	超级用户	
gongan	公安	正常	公安	B端用户	
minzheng	民政	正常	民政	B端用户	
renshi	人事	正常	人事	B端用户	
laodong	劳动	正常	劳动和社会保障	B端用户	
renkou	人口	正常	人口计生	B端用户	
jiaoyu	教育	正常	教育	B端用户	
shuiwu	税务	正常	税务	B端用户	
weisheng	卫生	正常	卫生	B端用户	

« 1 » 共1页，转到第 1 页 Go

图 6-22 部门管理示意图

5. 审批事项管理

支持新增审批事项、查询审批事项清单、审批、查询审批事项详情。

审批事项管理示意图如图 6-23 所示。

审批事项列表

事项名称	审批类型	事项种类	发起人	发起时间	事项状态	操作
发布合约: deployChainCode[PointChaincode:1.0]	deployChainCode	一票否决	admin	2017-09-13 07:29:37	已通过	处理 交易
发布合约: deployChainCode[ListChaincode:1.0]	deployChainCode	一票否决	admin	2017-09-13 07:29:19	已通过	处理 交易
发布合约: deployChainCode[ELicenseChaincode:1.0]	deployChainCode	一票否决	admin	2017-09-13 07:29:03	已通过	处理 交易
发布合约: deployChainCode[DictChaincode:1.0]	deployChainCode	一票否决	admin	2017-09-13 07:28:46	已通过	处理 交易
发布合约: deployChainCode[CryptChaincode:1.0]	deployChainCode	一票否决	admin	2017-09-13 07:28:31	已通过	处理 交易
发布合约: deployChainCode[AccessChaincode:1.0]	deployChainCode	一票否决	admin	2017-09-13 07:28:14	已通过	处理 交易

« 1 » 共1页，转到第 1 页 Go

图 6-23 审批事项管理示意图

仅供非商业用途或交流学习使用

6. 会员积分管理

支持会员积分查询、更改、积分变化历史记录查询功能。

会员积分管理示意图如图 6-24 所示。

会员列表　请输入会员编号　查询

编号	名称	状态	积分	操作
shuiwu	税务	正常	1687	查看记录 修改积分
weisheng	卫生	正常	1381	查看记录 修改积分
laodong	劳动	正常	1879	查看记录 修改积分
renshi	人事	正常	1426	查看记录 修改积分
gongan	公安	正常	1090	查看记录 修改积分
minzheng	民政	正常	1921	查看记录 修改积分

« 1 » 共1页，转到第 1 页 Go

图 6-24　会员积分管理示意图

历史积分查看示意图如图 6-25 所示。

shuiwu积分历史

主体ID	证照编号	证照类型	流水号	操作	操作类型	积分	时间
440301505294512932	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25
440301505294512884	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:24
440301505294512154	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:24
440301505294512188	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:24
440301505294512388	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25
440301505294512382	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25
440301505294512323	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25
440301505294512404	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25
440301505294512476	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25
440301505294512470	NJ02QT01-005	列表型	NJ02QT022017042514141400005	上传新证照	增加	3	2017-09-13 17:18:25

« 1 2 3 4 5 23 » 共23页，转到第 1 页 Go

图 6-25　历史积分查看示意图

6.9.4　系统管理功能

1. 用户管理

实现系统用户的增加、删除、修改、查询。

仅供非商业用途或交流学习使用

2. 角色管理

实现系统角色的增加、删除、修改、查询。实现角色的权限分配。

角色管理示意图如图 6-26 所示。

图 6-26　角色管理示意图

3. 日志、告警、系统状态监控

实现查询系统操作日志、查询系统业务日志、查看日志文件等功能；实现系统告警的查询和处理；实现系统状态监控，包括 CPU、内存、磁盘、业务统计数据等关键指标的监控。

系统监控的示意图如图 6-27 所示。

图 6-27　系统监控示意图

仅供非商业用途或交流学习使用

6.10　智能合约设计

6.10.1　智能合约多层结构设计

从业务相关流程考虑，智能合约设计包括对政务服务数据的查询和修改。这里采用多层结构、松耦合的方式，把业务与底层分离，易于扩展，增加新业务，同时保证数据的可靠和稳定。根据功能可以将智能合约模块内部划分为若干功能块：a 区块链访问层、b 标准访问 client 层、c 会员层等。本案例的设计实践上，智能合约模块具有 a-b-c-d-e-Entity 多层结构，用户定义的智能合约为 d'，通过实现 b 层的 client 而对区块链底层的数据进行操作。e.manager 则对智能合约模块进行管理，Entity 层是支撑智能合约的实体层。各层模块划分与功能结构图如图 6-28 所示。

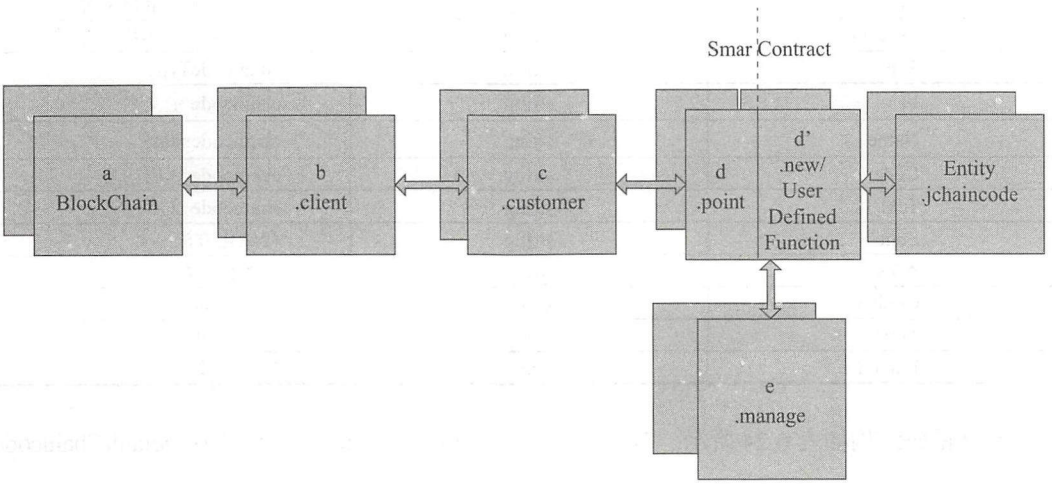


图 6-28　智能合约模块划分与功能结构图

各层分别介绍如下。

1）blockchain.b.client：连接 Blockchain 区块链环境的操作对象集合，包含节点管理、加解密与签名处理、交易执行、MemberService 访问的接口 Client 操作对象。

2）blockchain.c.customer：基于 blockchaincustomer 合约的操作方法，封装的会员注册、查询、修改、更换证书等操作方法。

3）blockchain.d.point：基于 PointAccount 合约的操作方法，封装的面向积分的发行授权、转账、查询、交易记录查看等方法。

4）blockchain.e.manage：基于 Blockchain 的操作对象实现的对区块链平台的管理功能，包括合约管理、会员管理、账本浏览等功能。

5）Entity.jchaincode：Java 智能合约的基础类支持。注意该模块如果发生修改必须将代码合并至每个 fabric/core/chaincode/shim/java 的路径下，并对 Fabric 进行 make all 动作，生成新的 javaenv 的 docker image。

6.10.2 智能合约模块设计

智能合约模块可以根据具体业务需求实现相应业务功能。这里以几个典型的智能合约为例进行说明。

1. 管理合约设计

该合约主要实现对其他合约的管理，包括注册、升级等功能。

（1）数据模型设计

链码结构表如表 6-23 所示。

表 6-23 链码结构表

属 性	类 型	说 明
Id	string	Id（Id+ 版本组合主键）
Version	string	版本（Id+ 版本组合主键）
Type	string	chaincodeType
Id	string	chaincode 中文名
Name	string	chaincodename
Code	string	chaincode 代码内容
Text	string	chaincode 文本描述
Func	string	初始化方法
Args	string	初始化参数
Creater	string	发布人 Id
Time	string	注册时间
Extend	string	扩展属性

默认链码结构如表 6-24 所示。对应表名：DefaultChaincode，对应结构：defaultChaincode。

表 6-24 默认链码结构

属 性	类 型	说 明
Id	string	Id（主键）
Version	string	版本
Type	string	chaincodeType
Id	string	chaincode 中文名
Name	string	chaincodename
Code	string	chaincode 代码内容
Text	string	chaincode 文本描述
Func	string	初始化方法
Args	string	初始化参数
Creater	string	发布人 Id
Time	string	注册时间
Extend	string	扩展属性

（2）接口列表

1）deploy 方法接口如下。

init：Controller 链码发布，初始化表结构

2）invoke 方法接口如下。

register：注册链码版本到 Controller

setDefault：设置链码默认版本

copyState：复制智能合约数据到新的智能合约上

3）query 方法接口如下。

getChainCodeName：根据链码 Id 获取默认链码 Name

getDefault：根据链码 Id 获取默认链码详细信息

getDefaults：获取所有的默认链码版本列表

getChaincode：根据链码 Id、版本号获取链码详细信息

getChaincodes：获取所有链码版本列表

getChaincodeVersions：根据链码 Id 获取此链码的所有版本列表

（3）接口设计

1）deploy-init，Controller 链码发布，初始化表结构参数说明如表 6-25 所示。

表 6-25　初始化表结构

请求参数说明		
Method	deploy	请求类型
Cert	用户 cert	
Signature	用户消息签名	
Path	Controller 所在路径	
Function	init	
Args	[]	

2）invoke-register，注册链码版本到 Controller。注册链码版本到 Controller 参数说明如表 6-26 所示。

表 6-26　注册链码版本到 Controller

请求参数说明		
Method	invoke	请求类型
Cert	用户 cert	
Signature	用户消息签名	
ChaincodeName	deploy 生成的 chaincodename	
Function	register	
Args	[id,version,code,text,name,time,extend]	id：别名 version：版本 code：链码 text：链码文本 name：链码名 time：注册时间 extend：扩展

3）invoke-setDefault，设置链码默认版本，如表 6-27 所示。此接口只能通过 approval 来调用。

表 6-27 设置链码默认版本

Method	invoke
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	setDefault
Args	[id,version,time]

4）invoke-copyState，复制智能合约数据到新的智能合约。请求参数表如表 6-28 所示。

表 6-28 请求参数表

Method	invoke
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	copyState
Args	[sourcechaincodename, destchaincodename]

5）query-getChainCodeName，根据链码别名获取默认链码名。请求参数表如表 6-29 所示。

表 6-29 请求参数表

Method	query
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	getChainCodeName
Args	[id]

6）query-getDefault，根据链码别名获取默认链码详细信息。请求参数表如表 6-30 所示。

表 6-30 请求参数表

Method	query
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	getDefault
Args	[id]

7）query-getDefaults，获取所有的默认链码版本列表。请求参数表如表 6-31 所示。

表 6-31　请求参数表

Method	Query
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	getDefaults
Args	[]

8）query-getChaincode，根据链码别名、版本号获取链码详细信息。请求参数表，如表 6-32 所示。

表 6-32　请求参数表

Method	query
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	getChaincode
Args	[id,version]

9）query-getChaincodes，获取所有链码版本列表，如表 6-33 所示。

表 6-33　获取所有链码版本列表

Method	Query
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	getChaincodes
Args	[]

10）query-getChaincodeVersions，根据链码别名获取此链码的所有版本列表，如表 6-34 所示。

表 6-34　根据链码别名获取此链码的所有版本列表

Method	query
Cert	用户 cert
Signature	用户消息签名
ChaincodeName	deploy 生成的 chaincodename
Function	getChaincodeVersions
Args	[id]

2. 联合审批合约设计

该合约主要实现在业务过程中需要查询待审批的事项，并完成审批的业务功能。

（1）数据模型设计

审批事项如表 6-35 所示。

表 6-35 审批事项

属 性	类 型	说 明
Id	string`json:"Id"`	Id（主键）
Name	string`json:"Name"`	审批事项
Type	string`json:"Type"`	审批事项类型（1：版本升级）
VetoType	string`json:"VetoType"`	事项联合审批类型（1：一票否决，2：2/3 同意，3：半数同意）
Chaincode	string`json:"Chaincode"`	执行的智能合约
Funtion	string`json:"Funtion"`	执行合约的方法
Args	string`json:"Args"`	执行方法的参数
Creater	string`json:"Creater"`	事项发起人会员号
Time	string`json:"Time"`	发起时间
Status	string`json:"Status"`	事项状态（1：审批中，2：审批通过，3：审批不通过）
Signs	[]*Sign`json:"Signs"`	会员签名
Extend	string `json:"Extend"`	扩展属性

会员签名如表 6-36 所示。

表 6-36 会员签名

属 性	类 型	说 明
CustomerNo	string`json:"CustomerNo"`	审批人会员号
Status	string`json:"Status"`	审批人是否同意（1：同意，2：不同意）
Time	string`json:"Time"`	审批时间

（2）接口列表

1）deploy 方法接口如下。

init：approval 链码发布

2）invoke 方法接口如下。

newApproval：新建联合审批事项

approval：联合审批

3）query 方法接口如下。

getApproval：根据联合审批事项 Id 获取联合审批事项详细

getApprovals：获取联合审批事项列表

（3）接口设计

1）deploy-init：approval 链码发布，指定参数为智能合约的别名。请求参数表如表 6-37 所示。

表 6-37　请求参数表

Method	deploy
Cert	用户 cert
Signature	用户消息签名
Path	Controller 所在路径
Function	init
Args	[chaincodealias]

2）invoke-newApproval，新建联合审批事项。请求参数表如表 6-38 所示。如果只有一个超级会员，直接联合审批通过；其他根据联合审批类型进行判断设置。联合审批成功后，执行目标智能合约、目标方法。

表 6-38　请求参数表

Method	invoke	请求类型
Cert	用户 cert	
Signature	用户消息签名	
ChaincodeName	deploy 生成的 chaincodename	
Function	newApproval	
Args	[Id,Name,Type,VetoType,Chaincode,Funtion,Args,Time,Extend]	Id：事项 Id Name：事项名称 Type：事项类别 VetoType：联合审批类别 Chaincode：联合审批要执行的智能合约别名 Function：联合审批要执行的智能合约方法名 Args：联合审批要执行的智能合约方法参数 Time：创建时间 Extend：扩展属性

3）invoke-approval，联合审批。请求参数表如表 6-39 所示。只有超级会员可以联合审批，联合审批达成条件时，执行目标方法。

表 6-39　请求参数表

Method	invoke	请求类型
Cert	用户 cert	
Signature	用户消息签名	
ChaincodeName	deploy 生成的 chaincodename	
Function	approval	
Args	[Id,Status,Time]	Id：联合审批事项 Id status：是否同意（1：同意，2：不同意） time：联合审批时间

4）query-getApproval，根据联合审批事项 Id 获取联合审批事项详细信息。请求参数表如表 6-40 所示。

表 6-40 请求参数表

Method	query	请求类型
Cert	用户 cert	
Signature	用户消息签名	
ChaincodeName	deploy 生成的 chaincodename	
Function	getApproval	
Args	[id]	id：联合审批事项 id

5）query-getApprovals，获取联合审批事项的列表。请求参数表如表 6-41 所示。

表 6-41 请求参数表

Method	query	请求类型
Cert	用户 cert	
Signature	用户消息签名	
ChaincodeName	deploy 生成的 chaincodename	
Function	getDefaults	
Args	[pagenum]	Pagenum：页数，0 表示获取所有

6.10.3 智能合约二次开发

数据共享平台主要的价值在于简政、惠民、兴业。各地政府面向个人和企业的办事事项非常繁杂，一个城市梳理出来的政务服务事项会有几千种，包含行政许可、行政征收、行政给付、行政确认、其他事项几大类。某市梳理出来的一部分政务事项清单如图 6-29 所示。

事项名称	部门	辖区	事项类型
城市建筑垃圾处置核准	市城市管理局	市本级	行政许可
渣土运输企业市场准入	市城市管理局	市本级	行政许可
房屋建筑类工程建筑垃圾处置核准	市城市管理局	市本级	行政许可
水利交通等其他类工程建筑垃圾处置核准	市城市管理局	市本级	行政许可
市政基础设施类工程建筑垃圾处置核准	市城市管理局	市本级	行政许可
我省居民赴港澳通行证件及签注的许可	市公安局	市本级	行政许可
机动车驾驶人驾驶证遗失补证、机动车驾驶人信息发生变化换证或驾驶证损毁换证、机动车驾驶证延期审验	市公安局	市本级	行政许可
中国公民因私出国护照	市公安局	市本级	行政许可
大陆居民往来台湾通行证及签注的许可	市公安局	市本级	行政许可
省重点及省重点以下的野生动物及其产品经营利用核准	市绿化园林局	市本级	行政许可
城市绿化工程设计方案审批	市绿化园林局	市本级	行政许可
省重点保护和国家保护的“三有”陆生野生动物驯养繁殖许可	市绿化园林局	市本级	行政许可
城市园林绿化企业三级企业资质许可	市绿化园林局	市本级	行政许可
经营（含加工）木材许可	市绿化园林局	市本级	行政许可
城市树木砍伐、移植、大修剪审批（门坡、道口）	市绿化园林局	市本级	行政许可
城市树木砍伐、移植、大修剪审批（绿化改造调整（无红线变化））	市绿化园林局	市本级	行政许可
城市树木砍伐、移植、大修剪审批（严重影响相邻建筑物采光、通风、通行，对人身安全或者其他设施构成威胁的）	市绿化园林局	市本级	行政许可
城市树木砍伐、移植、大修剪审批（涉及公园景区的）	市绿化园林局	市本级	行政许可
城市树木砍伐、移植、大修剪审批（建设工程项目（地块开发、道路、隧桥、地铁、杆管线等））	市绿化园林局	市本级	行政许可
改变绿化规划、绿化用地的使用性质审批（涉及公园景区的）	市绿化园林局	市本级	行政许可
改变绿化规划、绿化用地的使用性质审批（绿化调整改造（无红线变化））	市绿化园林局	市本级	行政许可
改变绿化规划、绿化用地的使用性质审批（门坡、道口）	市绿化园林局	市本级	行政许可
改变绿化规划、绿化用地的使用性质审批（建设工程项目（地块开	市绿化园林局	市本级	行政许可

事项清单信息 +

在 1824 个记录中筛选出 1157 个

图 6-29 某市政务事项清单

政府对个人或企业的服务事项所涉及的法定程序和环节，以及时限要求（如申请、受理、审查、决定、制证发证等）进行规定和公开，并附以相应的文字说明。申请材料、办事流程、时限要求等完全公开。平台保存了用户的数据而且可信，基于平台内的数据和办事的要求制定相应的智能合约，通过程序化的过程保障办事高效、透明、公开。然而智能合约是需要专业程序开发人员才能编写的，门槛较高，为了让政府办事人员能根据办事规则设计出智能合约，提供可视化的智能合约设计环境变得极具价值。以下为两个智能合约的可视化编排的示例。

新生儿入户智能合约可视化编排如图 6-30 所示。

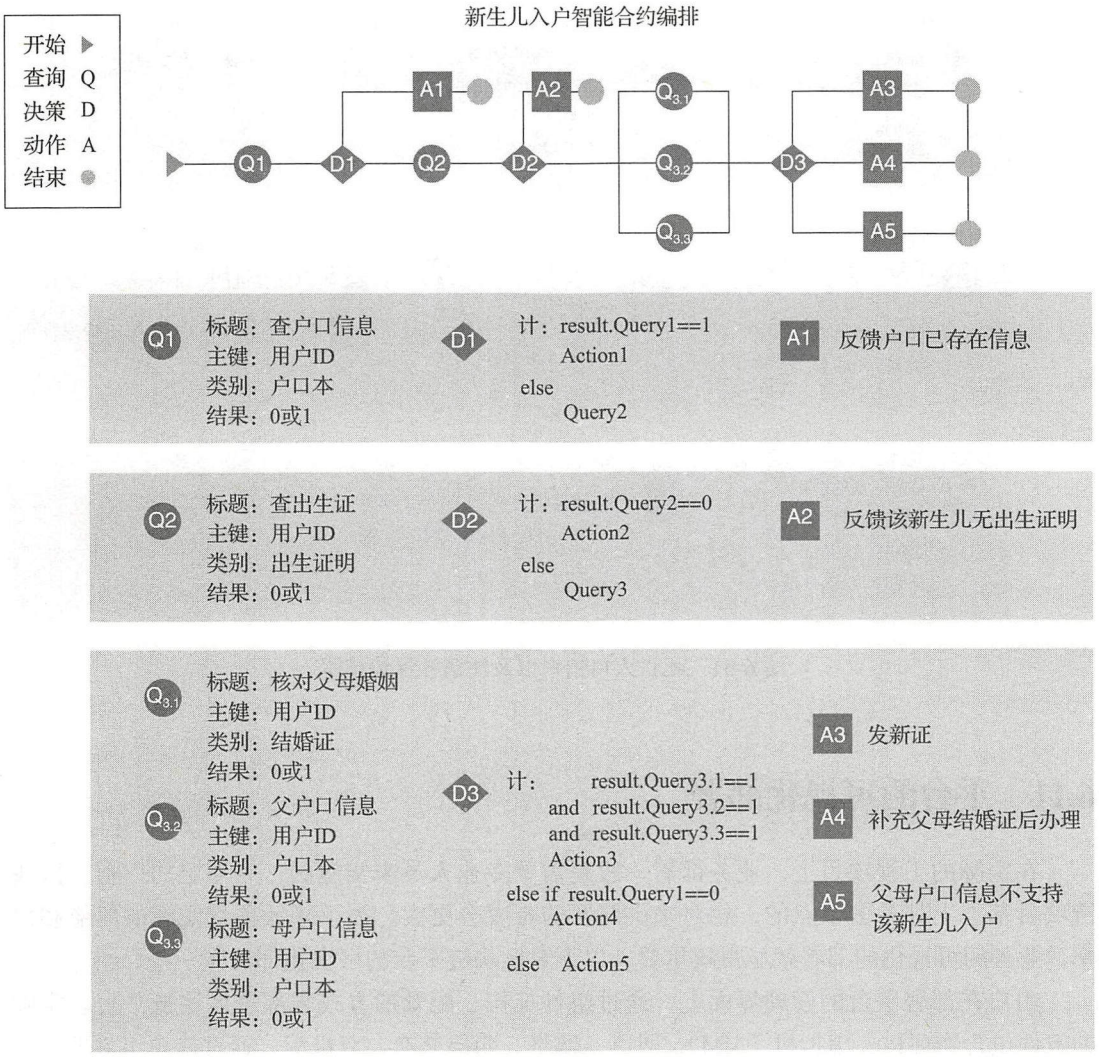


图 6-30　新生儿入户智能合约编排

死亡人口销户以及注销社保编排如图 6-31 所示。

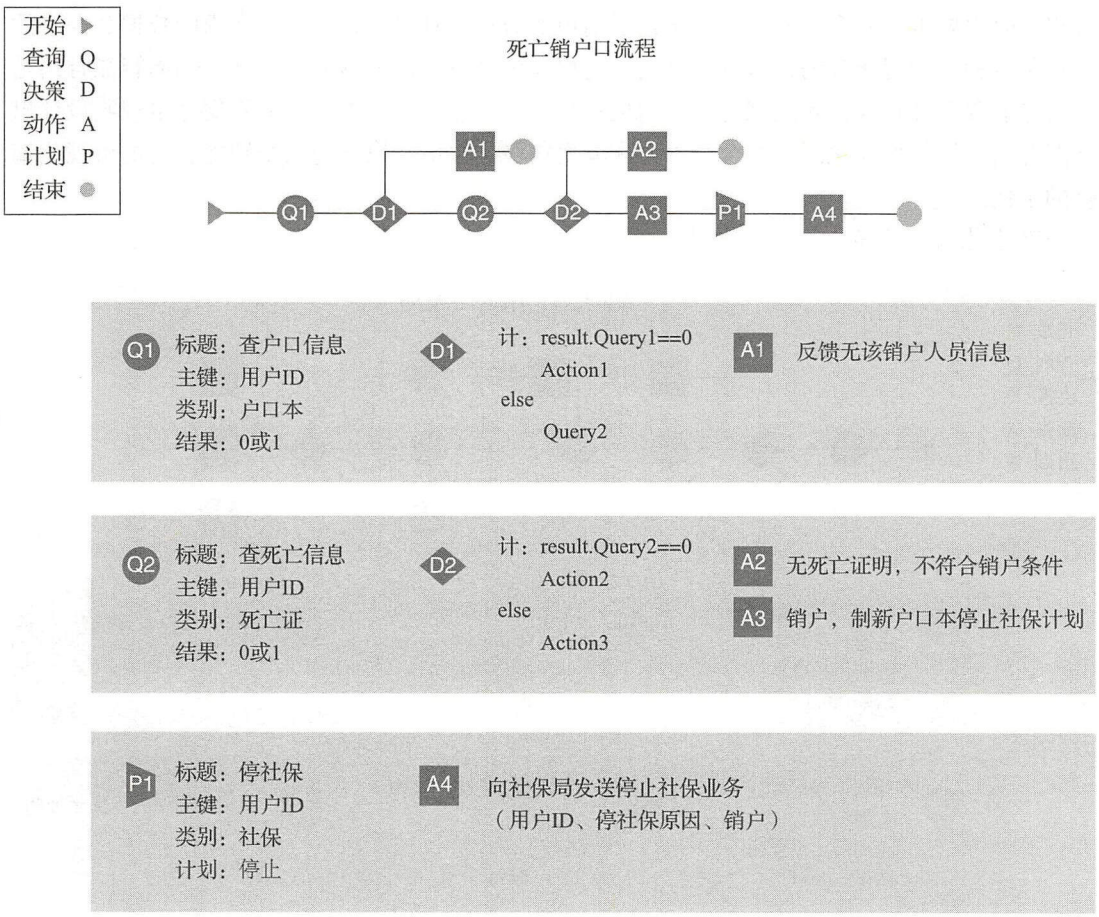


图 6-31 死亡人口销户以及注销社保编排图

6.11 平台的可视化部署

在实际的工程项目上，业务部署一般是需要专业人员来实施的，有一定的门槛。区块链项目基本上都是多节点的，相对来讲部署的难度会更大。为了降低项目实施的门槛和效率，业务的可视化的部署就显得很重要。以下对区块链平台的可视化部署做一些介绍。

用户在部署平台的管理界面上，通过组件拖拉、配置的方式来部署区块链平台。实际的组件包含物理机、虚拟机（VM）、网口、网络、四层交换、双机等，通过拖拉组件即可初步搭建区块链平台，如图 6-32 所示。

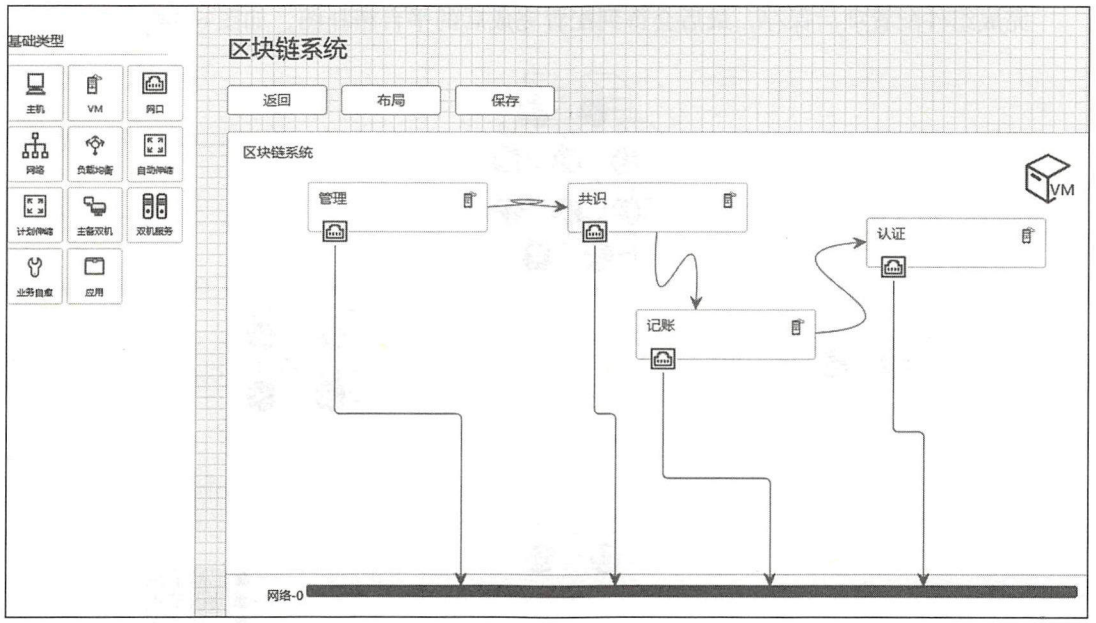


图 6-32　可视化部署组件

完成基础组件搭建后，对每个组件进行业务配置，比如设置组件的类型、安全、镜像、规格等，如图 6-33 所示。

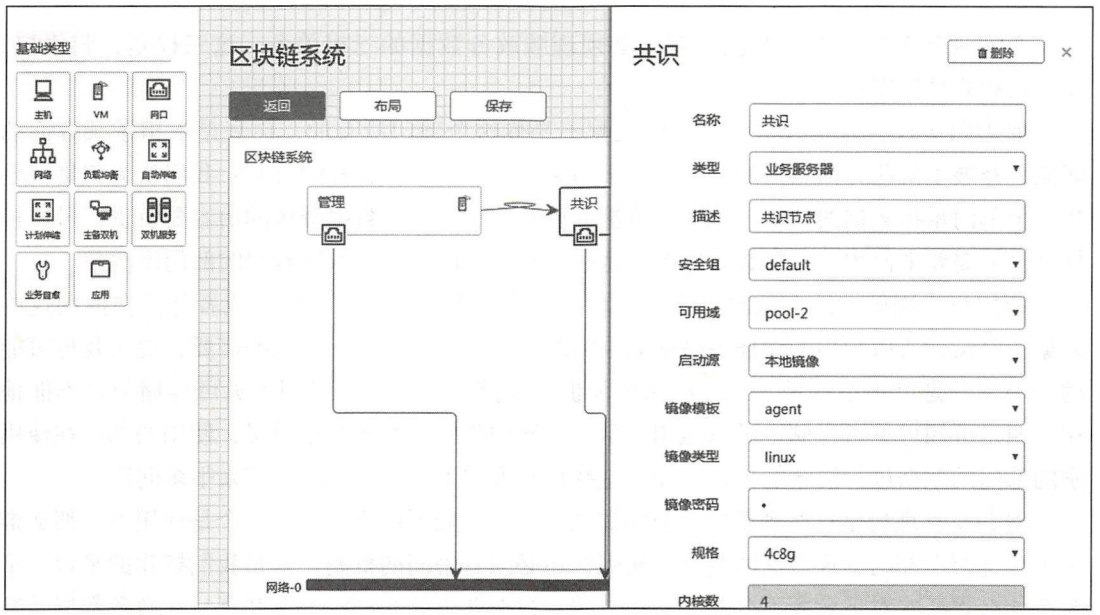


图 6-33　组件配置

完成组件配置项后保存，最后会得到配置完成后的平台拓扑图，如图 6-34 所示。

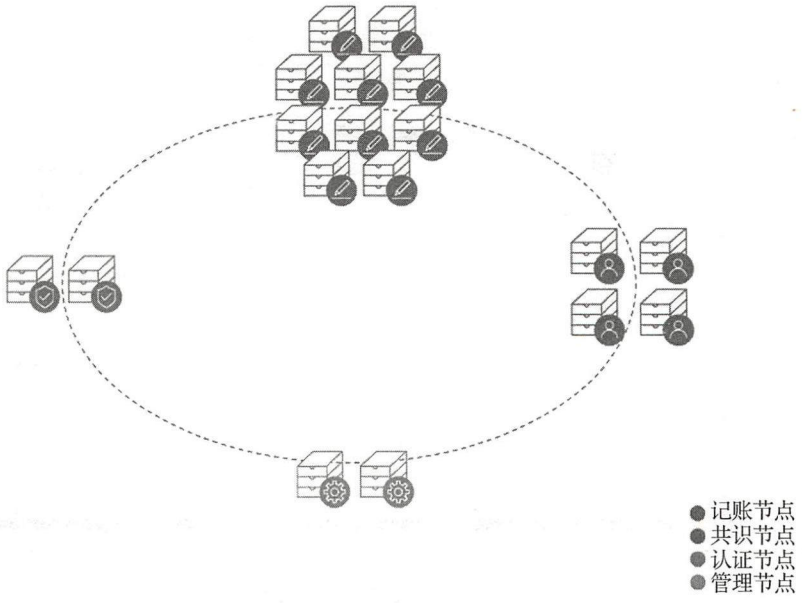


图 6-34 平台拓扑图

6.12 政务数据的三权关系

为实现政务数据共享交换，首先必须理顺政务数据的三权关系。这三权是，归集权、使用权和共享管理权。

何为归集权？就是说这些部门的数据不是你这个部门所有的，你只是承担了某方面的职能，开展了某些方面数据的归集。基于归集，应该给予什么样的权利？我们认为归集权是指一个部门承担数据的搜集、整理、维护、更新等工作，对数据资源的内容要负责，可以根据自身的需要来使用、支配数据资源。这些权利应该向拥有数据所有权的部门说清楚。

使用权好理解，这个数据到交换平台上，或者拿出来以后，机构或者部门在使用这些数据的时候，所应承担的责任不是毫无限制的。比如，我们基于我们的需要，提出几种可能的分类，一是可用不可见，二是可用可见但是不能存储，三是可用可见可存储但是不能他用，四是可用可见可存储也可以他用。基于不同的数据类型和数据效果，使用的部门在使用别的部门的数据进行清晰的界定。如果这些权利界定不清楚，可能会带来很多问题。

对共享管理权是这么理解的：前面提到了一个是数据的贡献方，一个是使用方。那么谁来主导这两个部门实现数据的共享交换呢？还必须有中间的机制，就是我们搭建的平台。平台有什么责任？对平台要不要共享交换？政务数据能不能共享？怎么共享？在政务数据共享过程中，应该由拥有共享管理权的部门深入调研，对数据共享中出现的争议问题进行处置和

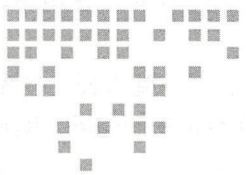
裁决，并赋予相应的机制这种权利。比如对于不按规则存储、维护、使用的数据部门进行责任的追溯。

关于数据三权更详细的论述，可参考清华大学公共管理学院教授、国家数据研究院执行院长孟庆国于 2018 年 5 月 26 日在政务信息系统整合共享论坛的演讲，网址如下：

http://zb.cbdio.com/308/index.html?from=groupmessage

区块链依据一种智能合约，我们把这些约定、这些权责关系通过程序的方式建立一个自动化的一个界定机制，来实现基于数据共享管理权的共享交换。这种机制如果能够建立起来的话，那么，无论前面提到的那四种分类中的哪一种都可以能够很好地实现。比如，具体哪一项业务需要可见、可用，我们要对它进行一个约定。也可以事先约定好，通过智能合约的方式实现数据的整合共享。

基于这三权，建立职责清晰、可控、可信、可追溯的一个政务数据整合交换机制。基于区块链和智能合约技术打造这样一个平台是一个创新，探索数据的三权创新共享机制，建立权责清晰、可信、可溯的共享交换机制，对拓展共享及创建共享机制、拓展共享思路很有启发意义。



Chapter 7 第 7 章

区块链应用设计

7.1 区块链在数字商票中的应用

7.1.1 简述

所谓数字商票，并不是新产生的一种实物票据，也不是单纯的虚拟信息流，它是应用区块链技术，结合现有的票据属性、法规和市场，开发出的一种全新的票据展现形式，与现有的电子票据相比在技术架构上完全不同。同时，它既具备电子票据所有功能和优点的基础，又融合了区块链技术的优势，是一种更安全、更智能、更便捷、更具前景的票据形态。

7.1.2 区块链解决的关键问题

（1）对于发行方来说，希望控制交易风险

解决行业发展多中心化发展趋势导致的除央行 ECDS 之外，纸质票据仍然占有很大比例的情况；解决传统商票票据真伪、一票多卖、背书不连续等问题。

（2）对于交易方来说，希望提升交易效率并降低交易成本

解决现实票据价值传递存在中介化的问题；希望实现发行方与交易方价值的限定与流转方向的控制。

（3）对于监管方来说，希望增强交易监管

解决信息的记载真实性与溯源问题；实现快速审查与调阅。

7.1.3 方案描述

1. 组网模型

组网模型如图 7-1 所示。

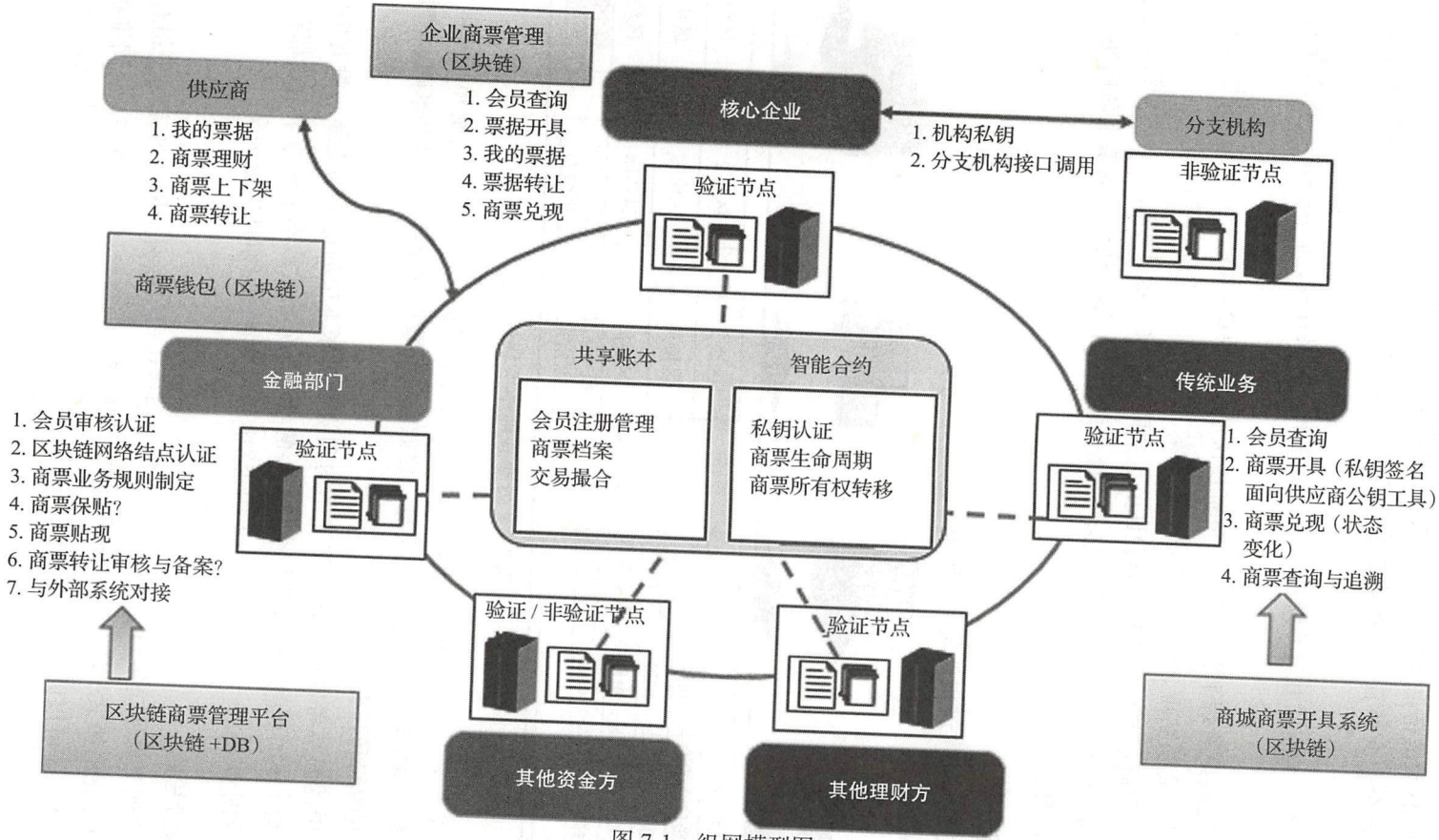

企业商票管理
（区块链）
1. 会员查询
2. 票据开具
3. 我的票据
4. 票据转让
5. 商票兑现
供应商
1. 我的票据
2. 商票理财
3. 商票上下架
4. 商票转让
核心企业
1. 机构私钥
2. 分支机构接口调用
分支机构
非验证节点
验证节点
商票钱包（区块链）
共享账本
智能合约
会员注册管理
商票档案
交易撮合
私钥认证
商票生命周期
商票所有权转移
金融部门
传统业务
1. 会员审核认证
2. 区块链网络结点认证
3. 商票业务规则制定
4. 商票保贴?
5. 商票贴现
6. 商票转让审核与备案?
7. 与外部系统对接
验证节点
验证节点
1. 会员查询
2. 商票开具（私钥签名面向供应商公钥工具）
3. 商票兑现（状态变化）
4. 商票查询与追溯
验证 / 非验证节点
验证节点
区块链商票管理平台
（区块链 +DB）
商城商票开具系统
（区块链）
其他资金方
其他理财方

图 7-1　组网模型图

系统共包括区块链企业商票管理、区块链商城商票开具系统、区块链商票管理平台三大部分。

1）区块链企业商票管理系统提供会员查询、票据开具、我的票据、票据转让、商票兑现等业务。

2）区块链商城商票开具系统提供了会员查询、商票开具、商票兑现、商票查询与追溯等业务。

3）区块链商票管理平台提供了会员审核认证、区块链网络节点认证、商票业务规则制定、商票保贴、商票贴现、商票转让审核与备案、与外部系统对接等业务。

所有参与方通过各自的验证 / 非验证节点共同维护唯一的共享账本。共享账本记录了会员注管理、商票档案、交易撮合等信息；智能合约记录了私钥认证、商票生命周期、商票所有权转移等信息。

2. 实现过程

分布式账本实现过程如图 7-2 所示

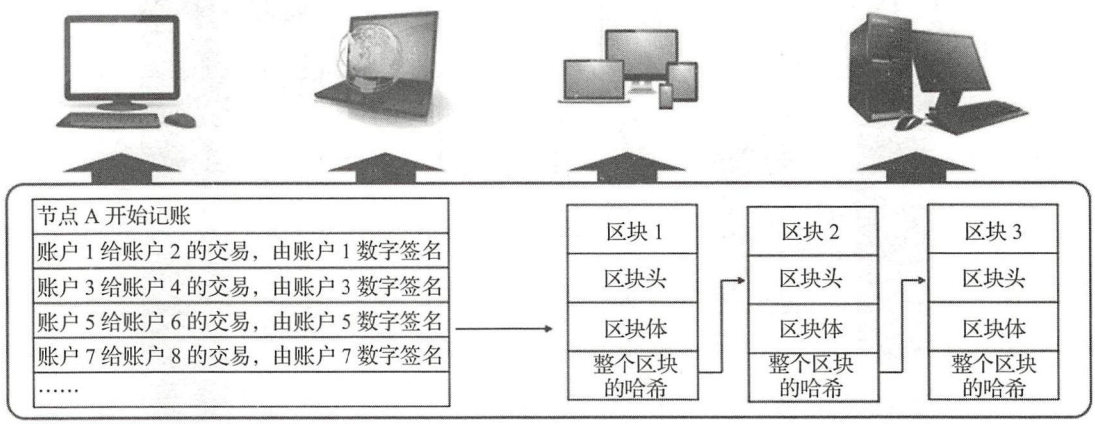


图 7-2　分布式账本实现过程图

分布式账本（Distributed ledger）是一种在网络成员之间共享、复制和同步的数据库。分布式账本记录网络参与者之间的交易，比如资产或数据的交换。这种共享账本消除了调解不同账本的时间和开支。

网络中的参与者根据共识算法来制约和协商对账本中的记录的更新。没有中间的第三方仲裁机构（比如金融机构或票据交换所）的参与。分布式账本中的每条记录都有一个时间戳和唯一的密码签名，这使得账本成为网络中所有交易的可审计历史记录。

3. 设计思路

1）票据交易所、银行、保险、基金等金融机构可以联合组网，各家处于相对平权（相比传统中心化的模式）的位置。在平台中，记账节点（高信用背书特征）共同维护联盟链，普通节点经认证可以同步联盟链的数据并使用这些数据，交易对手和交易过程被完整记录在联盟链上，不可篡改；平台会员以及平台上流转的资产一旦上链，就转化为可信状态且为所

有会员可见，避免了不同会员间重复 KYC 流程，大幅度降低了交易成本；而智能合约则进一步提高了交易的灵活性，降低了确认、清算和结算的成本。

2）设立一个身份管理机构，负责识别参与方身份；设立了数字票据交易平台的参与方门槛，解决了传统交易平台上中介横行的困境。这个用户身份的管理机构主要提供参与方身份的证书颁发、存储、验证、授权以及丢失恢复的服务。参与方在票据交易平台中进行登录、交易、查询等业务操作时，使用私钥进行认证与数据加密。

3）使用区块链承载数字票据的完整生命周期，并采用智能合约优化票据交易与结算流程，提高交易效率，并可以创造出很多全新的业务场景。数字票据业务系统包含开票、企业间流转、贴现、转贴现、再贴现、回购等一系列业务类型，这些业务类型以及交易中的要求和限制，都可以通过智能合约编程的方式来实现，并可根据业务需求变化灵活地变更或升级。票据交易智能合约可以自动完成资金转移、保证金锁定、手续费扣除、所有权的变更等票据交易动作。

4）利用区块链大数据与智能合约，实现票据交易的事中监管，降低监管成本。得益于区块链的技术特性，监管方可以随时对分布式账本的交易记录进行审计，而不需要依赖于票据交易平台所提供的接口。监管机构可以根据监管要求，设计开发用于监管的智能合约，并发布到数字货币区块链上，由票据交易智能合约在执行时作为前置合约进行调用，可以直接中止不符合监管要求的交易。每个交易的监管执行结果也会记录在区块链账本上。

智能合约是一种旨在以信息化方式传播、验证或执行合同的计算机协议。智能合约允许在没有第三方的情况下进行可信交易，这些交易可追踪且不可逆转。区块链上的所有用户都可以看到基于区块链的智能合约。

4. KYC 会员处理机制

KYC（Know your customer）是指对账户持有人的强化审查，是反洗钱及用于预防腐败的制度基础。可以了解资金来源合法性。

KYC 政策不仅要求金融机构实行账户实名制，了解账户的实际控制人和交易的实际收益人，还要求对客户的身份、常住地址或企业所从事的业务进行充分的了解，并采取相应的措施。

其特点如下：

1）金融机构和监管机构均加入区块链网络，每个机构作为区块链网络中一个独立节点；

2）金融机构采集和认证用户 KYC 信息，通过区块链分布存储到各个节点中，区块链可以确保 KYC 信息从采集到每次变更的可追溯和可验证，并有机构的签名确认；

3）金融机构发起的金融交易也需要区块链实时同步到监管机构节点，监管机构可以对交易属性进行事中监管或者事后监管；

4）存储在区块链网络中的 KYC 认证信息可被联盟链上的节点使用，使用规则受到一定的访问权限的控制；

5）基于区块链技术的网络可确保网络节点安全稳定和容错机制，保证了 KYC 信息安全存储，且无单一节点故障。

KYC 会员处理机制如图 7-3 所示。

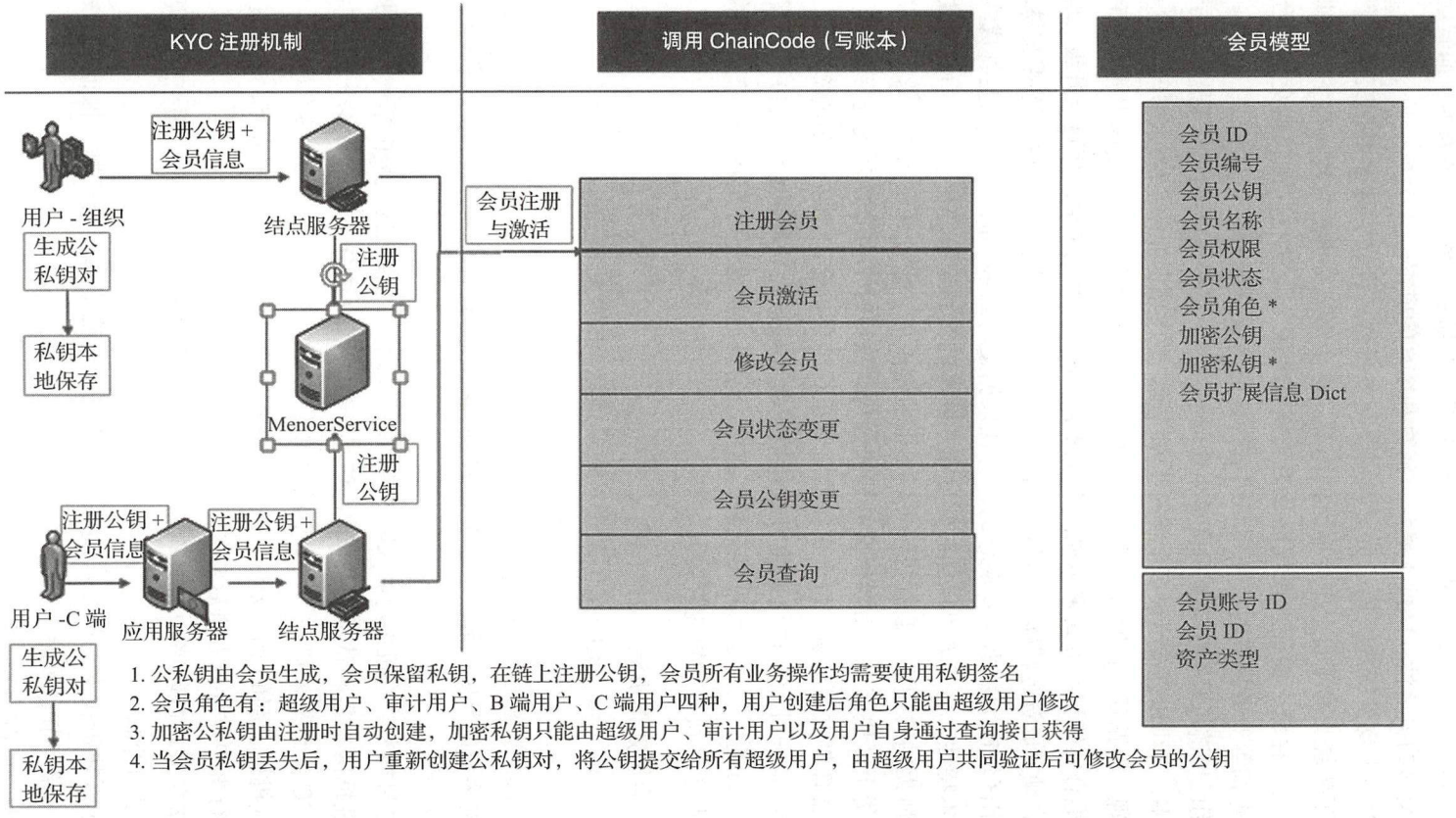


图 7-3　KYC 会员处理机制图

7.1.4　小结

1. 区块链数字商票总体特点

具有分布式、完整时间戳、不可篡改的特性。

1）不需要第三方对交易双方价值传递的信息做监督和验证。

2）纸质票据托管后形成的区块链票据。

3）联盟间认可的无纸化区块链数字票据。

2. 贯彻交易规则

1）参与各方可设定交易规则并自动执行，无人为干扰因素。

2）实时进行交易方信用搜集、评估与风险管控。

3. 提高交易效率，降低交易成本

1）区块链解决基础交易合法性问题，提升交易效率。

2）票据交易去中介化，三方变两方。

4. 监管方的深度参与

1）将监管方的法律法规作为基础规则，影响交易行为全过程。

2）通过区块链确保交易行为完整记录且可追溯。

7.2　区块链在文化交易中的应用

7.2.1　简述

文化产业现状

共享和零边际成本的新经济时代已经到来，文化产业生态正逐步去中心化或称非中心化、多中心化。由此，内容的生产、组织、管理、传播、消费、监管必然发生改变。

如图 7-4 所示，传统文化交易平台很多时候只能提供中心化的信用，以交易所作为担保，这就限定了交易往往是基于机构间、基于批发交易、基于标准化的产品。人与人之间的信用危机，导致个人与个人的交易是基于熟人或由中介完成。

而新兴文化产业交易正逐渐去中心化，它不再以交易所为中心，而是将交易所与经济、制作、消费、监管等环节并列，共同组成一个去中心化的文化产业交易平台，一起使用共识机制构建行业信用。

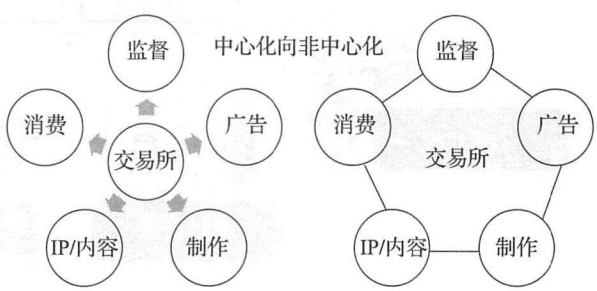


图 7-4　文化产业趋势图

7.2.2　区块链解决的关键问题

不管是对于监管部门还是生产者、

投资者、消费者来说，传统文化交易行业都存在着很多不足之处。

1）对于监管部门来说，版权保护无方法、监督管理无手段、品质内容无保证、传播行为无控制。

2）对于生产者、投资者、消费者来说，创意创造无舞台、共享协同无工具、权属利益不清晰、文化产品少精品。

文化交易总共包括 3 个参与方，即购买者、拥有者、监管者。三方各自有不同的需求。

1）对于购买者来说，希望看到好的文化产品，如果有，愿意为此支付费用。

2）对于拥有者来说，思想、创意和创造需要被保护、被尊重、被体现，希望付出能有所回报。

3）对于监管者来说，在互联网时代，需要一个新的管理工具和手段，推动行业持续健康发展。

这些诉求，在区块链技术产生之前没有很好的解决方案。

7.2.3 方案描述

1. 总体设计

（1）区块链文化交易平台节点设计

区块链文化交易平台节点图如图 7-5 所示。

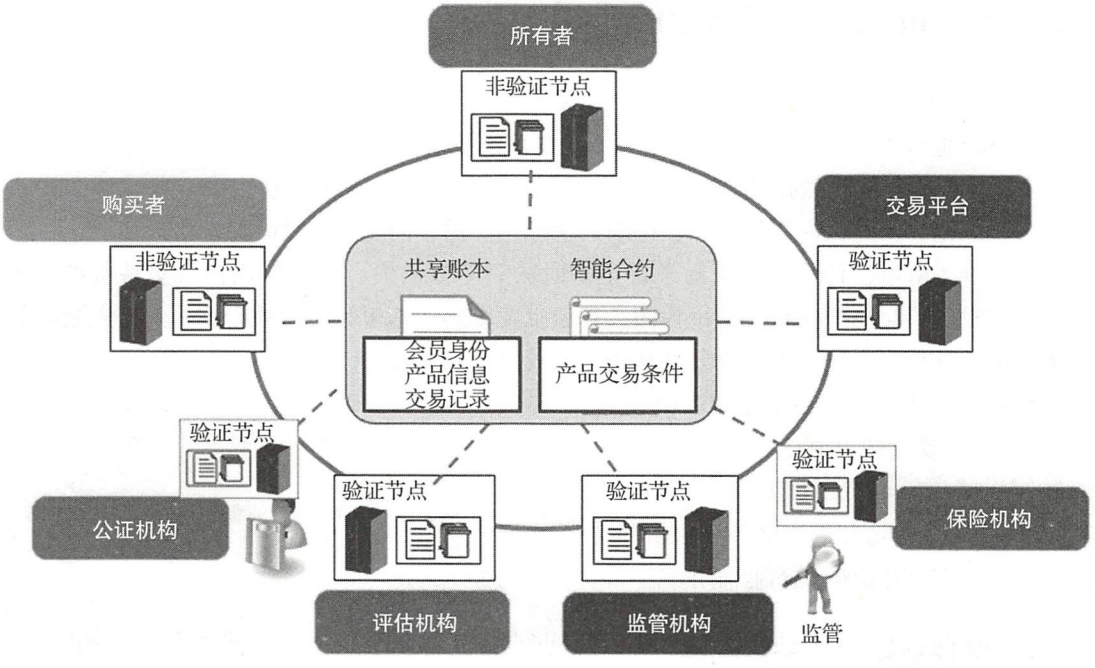


图 7-5 区块链文化交易平台节点图

区块链将资产进行鉴定并且将资产数字化，并将与该资产相关的所有行为均记录在账本上。文化交易的购买者与所有者通过非验证节点与共享账本 / 智能合约交换数据，公证机构、评估机构、监管机构、保险机构、交易平台等通过验证节点与共享账本 / 智能合约交换数据。

文化交易所有的参与方都共同维护唯一账本，同时智能合约提升了纸质合同的效率。

（2）区块链文化交易平台业务设计

区块链文化交易平台业务系统图如图 7-6 所示。

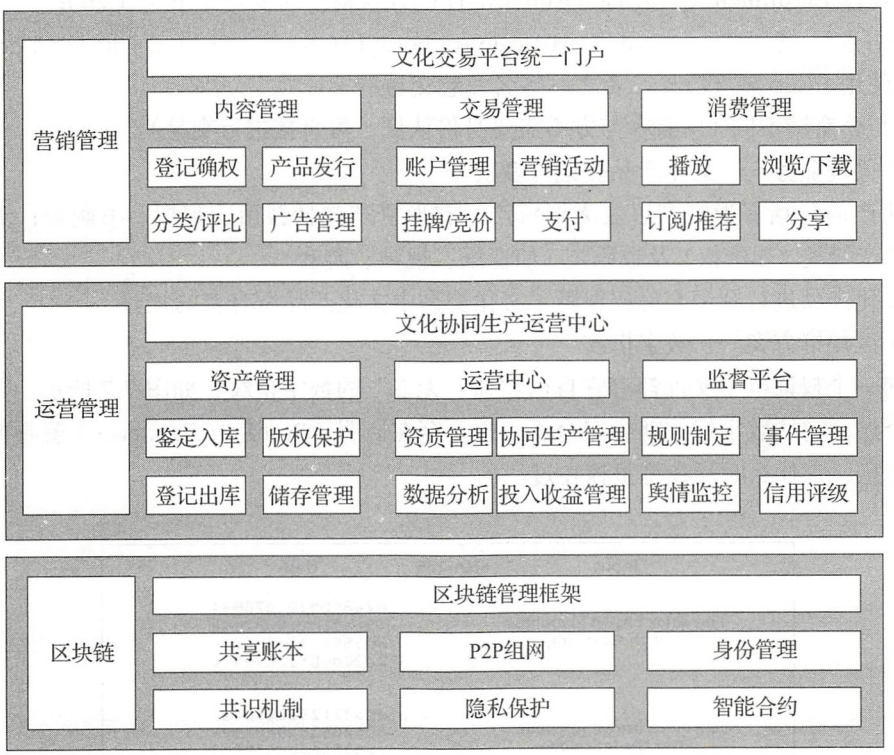


图 7-6　区块链文化交易平台业务系统图

底层为区块链管理框架，通过共享账本、P2P 组网、身份管理、共识机制、隐私保护、智能合约等技术为文化交易平台提供技术支持；中间层为文化协同生产运营中心，通过资产管理、运营中心、监督平台三大模块保障文化交易平台的正常运行；顶层为文化交易平台统一门户，提供面向用户的文化交易服务。

2. 关键业务的实现

区块链文化交易包含会员注册登记、文化产品登记与确权、交易与消费、监督等四大关键业务，这四大关键业务的成功实现是区块链文化交易成功的保证。

（1）关键业务——会员注册登记

会员注册登记主要包含会员信息与资质信息两部分。

1）会员信息：会员身份信息登记、会员账户信息、会员信息变更。

2）资质信息：工商税务信息、各类证照资质。

（2）区块链方案——成员管理

成员管理提供会员注册、身份保护、内容保密功能。

成员服务如下：

1）通过 Registration Authority(RA) 注册获得许可；

2）通过 Enrollment Certificate Authority(ECA) 获得注册安全证书（ECert）；

3）(可选)，通过 Transaction Certificate Authority(TCA) 获得交易安全证书（TCert）；

4）使用 ECert 或 TCert 二者之一签名发起交易请求；

5）与公有链不同（所有参与方不需要身份认证，可直接进行交易）。

（3）关键业务——文化产品登记与确权

文化产品包括有形资产以及无形资产。有形资产包括钱币、邮票、书画等；无形资产包括知识产权类（如版权）及数码类（如音乐、视频、游戏）。

对文化产品进行登记与确权保障了文化交易中文化产品所有者的权益。

（4）区块链方案——数字指纹

任何一个权证电子文件都有它自己“独一无二”的数字指纹，如图 7-7 所示。哪怕修改其中一个点，它的数字指纹就会完全不一样。存在证明（Proof Of Existence）就是要证明任何一段信息，它什么时候在哪里存在过。

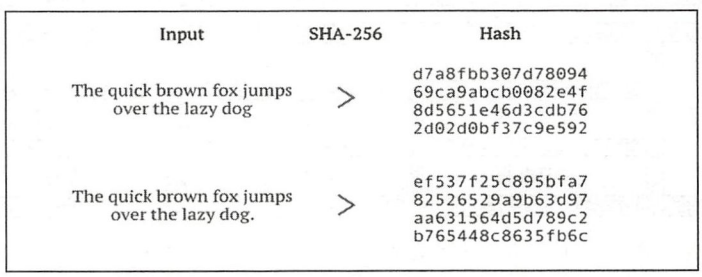


图 7-7 数字指纹

在二代区块链技术体系中，为每一个权证档案电子文件或交易指令提供一个数字指纹，编入文化产权交易所区块链，提供动态时间戳摘要，同时 Hash 指令执行记录，供查询中心查询文化产权证的数字指纹、区块链区块位点、变更记录等。通过区块链 POE 技术，构建权证可信基——数字指纹。

区块链在各个节点都有存储，数字指纹一旦生成，便永远无法被篡改或删除，任何改动都有时间戳记录在案，可以为监管部门提供可信追溯。

区块链数字指纹应用技术，为文化产权权证档案，提供了生命周期内的可信共享基础，无法被任何第三方机构篡改或删除。

（5）关键业务——交易与消费

交易与消费是文化交易的核心业务。购买者通过此业务买到自己心仪的产品，所有者通过此业务获取自己相应的报酬。

（6）区块链方案——智能合约

1）去信任交易。由于资产已经数字化，且均记录在区块链上，因此交易时不需要知道对方是谁，是否会存在欺诈。

2）非中心化交易。交易可以点对点的交易，交易由三方变为两方，也可以委托任意一家链上的交易所进行交易。

3）投入与收益。因为资产数字化后可以任意切割，因此投入和收益变得更加简单和明确。

（7）关键业务——监督

区块链（监督）系统图如图 7-8 所示。

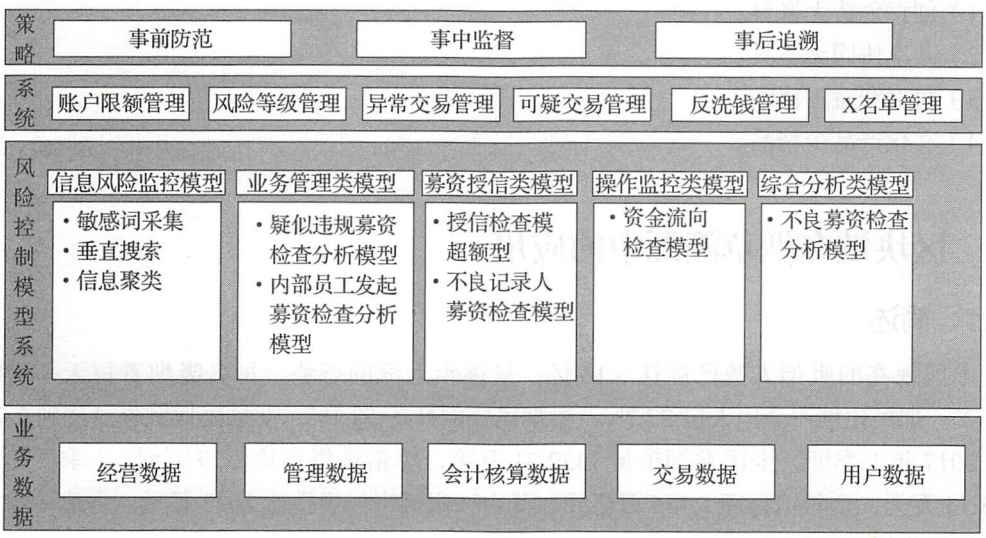


图 7-8　区块链（监督）系统图

底层为业务数据，包含经营数据、管理数据、会计核算数据、交易数据、用户数据等，为实现监督提供数据支撑。

第二层为风险控制模型系统，通过信息风险监控模型、业务管理类模型、募资授信类模型、操作监控类模型、综合分析类模型等为不同类型的监督提供相应的数据。

第三层为管理系统，包括账户限额管理、风险等级管理、异常交易管理、可疑交易管理、反洗钱管理、X 名单管理，此层给监管人员提供了不同的监管类型。

最顶层为监督策略，包括事前防范、事中监督、事后追溯。

区块链监督系统为监管人员提供了全方位的监管策略。

（8）区块链方案——追溯与审计

1）根据区块链不可篡改的特性，通过区块链确保交易行为完整记录且可追溯。

2）根据区块链可事先指定智能合约的交易规则，将监管方的法律法规作为基础规则，影响交易行为全过程。

7.2.4 小结

1. 解决了监管部门 4 大难题

1）版权保护无方法。

2）监督管理无手段。

3）品质内容无保证。

4）传播行为无控制。

2. 解决了生产、投资、消费领域 4 大痛点问题

1）创意创造无舞台。

2）共享协同无工具。

3）权属利益不清晰。

4）文化产品少精品。

7.3 区块链在烟草溯源中的应用

7.3.1 简述

中国现在的吸烟人数已高达 3.16 亿，呈逐年上涨的态势，每名吸烟者每天平均吸烟 15.2 支。烟民比例占全国人口 27.7%，男烟民比例高达 52.1%，女烟民比例也已达到 2.7%。

2017 年 1 季度，全国卷烟销量 1332.21 万箱，单箱销售均价 3.29 万元。1 季度销售额 4388.15 万元，全年销售额 1.755 万亿元。其中，假烟市场规模达 2633 亿元，占比 15%，国家损失税收超 800 亿元。

去年，全国共查获假烟 2.16 万件（总值 1.36 亿元），走私烟 1.18 万件，烟丝烟叶 873.1 吨，5 万元以上案件 279 起。

假烟的泛滥给国家税收带来了巨大损失，防治假烟成为目前烟草行业的当务之急。

7.3.2 区块链解决的关键问题

1. 投入大，收效微

1）投入大：新招层出，条形码、二维码、RFID 电子标签；高档烟甚至多重防伪技术并用。

2）收效微：现有防伪技术可复制性高，经营户是否售卖假烟、窜货目前主要依靠上门检查，效果很差。

仅供非商业用途或交流学习使用

2. 上报难，监管难

经销商与厂商之间互相推卸责任；监管部门职责不清，流程复杂，取证难。以上种种原因都间接导致了假烟泛滥，给国家税收和消费者自身利益都造成了损失。

7.3.3　方案描述

1. 组网图

系统由分布式节点构成，包括烟草专卖局节点、消费者节点、零售终端节点、市公司节点、省公司节点、卷烟厂节点以及索引节点，如图 7-9 所示。

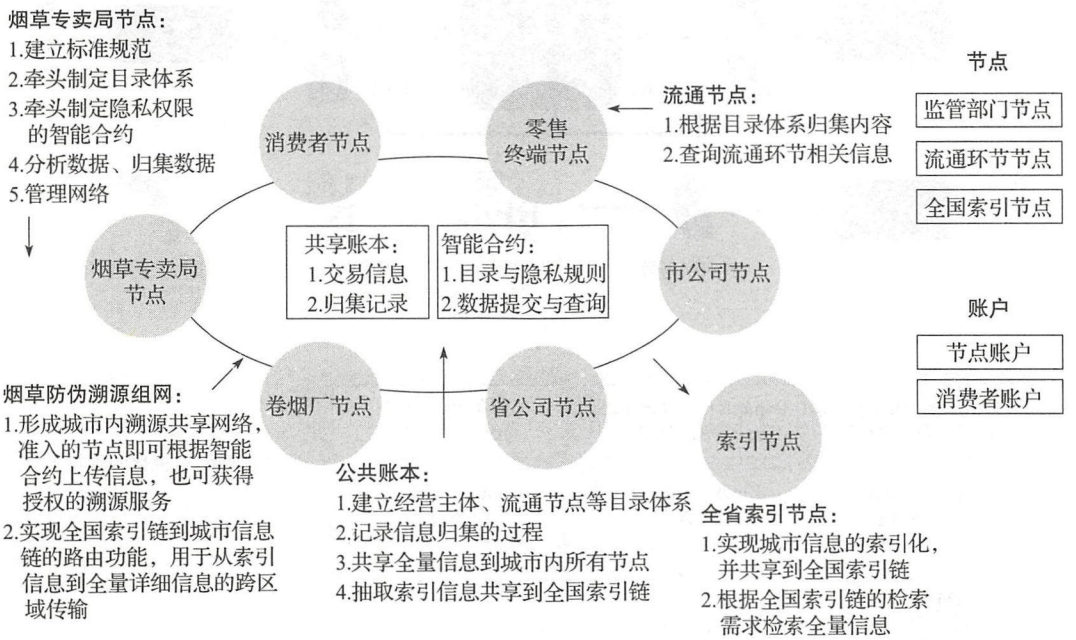


图 7-9　组网图

将交易信息以及收集记录等公布于共享账本中，供所有人查看；将目录与隐私规则、数据提交与查询等以编写智能合约的方式公布于对等网络上，实现多方业务组网。

2. 架构图

系统采用分布式架构。每个节点都部署节点服务器以及公共账本。业务系统以及终端设备（包括 RFID 标签、固定式读写器、手持式终端等）通过各个分布式节点获取数据，如图 7-10 所示。

3. 逻辑图

基于区块链的烟草防伪溯源总体架构共包含可视化的管控平台、联盟链核心功能、IT 基础设施三大部分，如图 7-11 所示。

仅供非商业用途或交流学习使用

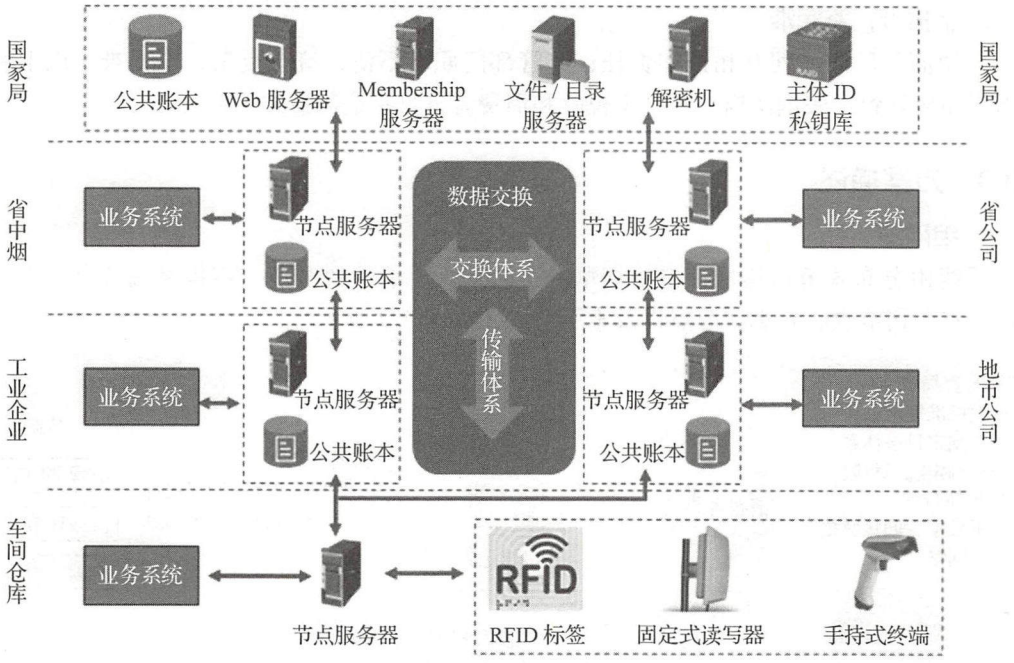


图 7-10 架构图

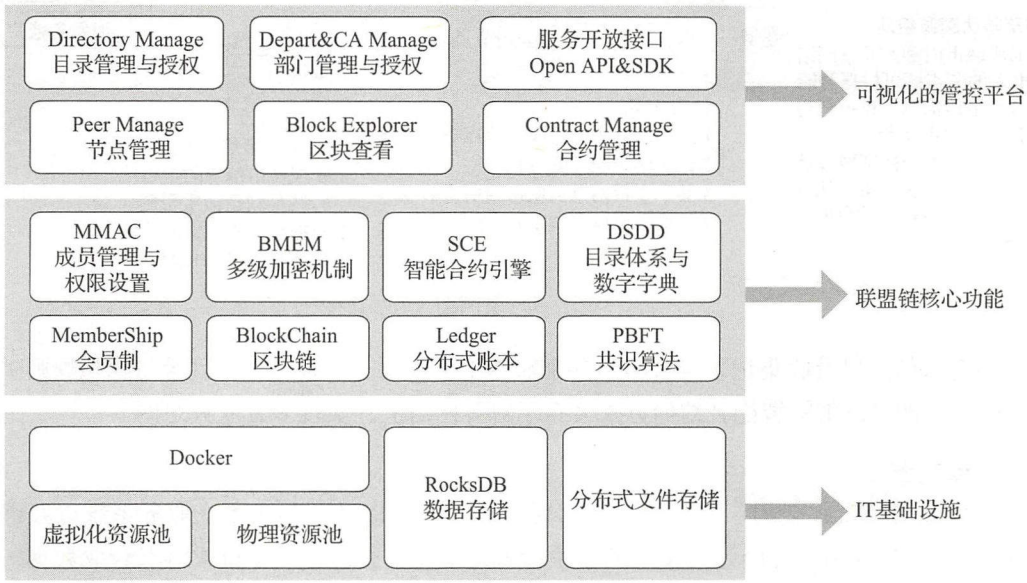


图 7-11 逻辑图

（1）可视化的管控平台

1）采用统一的云计算运维平台，提供功能强大的 Web 管理控制平台，灵活调整区块链部署，监测节点运行状态及区块信息。

仅供非商业用途或交流学习使用

2）标准化的 REST API/SDK/CLI 接口快速推动上层应用的开发，实现对区块链交易以及账本的快速写入、查询及历史浏览。

3）部门管理、目录体系管理与授权、智能合约的管理与部署等。

（2）联盟链核心功能

1）基于开源的 Hyperledger Fabric 架构，进行优化与改进。

2）由会员制、区块链、账本、共识算法、权限管理、多级加密、智能合约引擎、目录体系与数据字典等构成。

（3）IT 基础设施

1）在虚拟化资源池或物理资源池上构建 Docker 容器。

2）区块链底层存储基于 RocksDB/LevelDB。

3）分布式文件存储用于存放大文件。

4. 烟草防伪溯源思路

1）对于标准箱，采用工业级 RFID 芯片，根据频率不同，对大箱、小箱进行识别，存储箱子内件烟条码信息。

2）对于标准条，采用 RFID 芯片，将条烟信息和订单信息相关联，准确知道条烟何时何地销售给哪个零售户。

3）对于标准包，采用 OFID 芯片记录每包香烟的流通信息，防窜货、防伪。

5. 烟草防伪溯源系统数据流程图

烟草防伪溯源系统数据流程如图 7-12 所示。

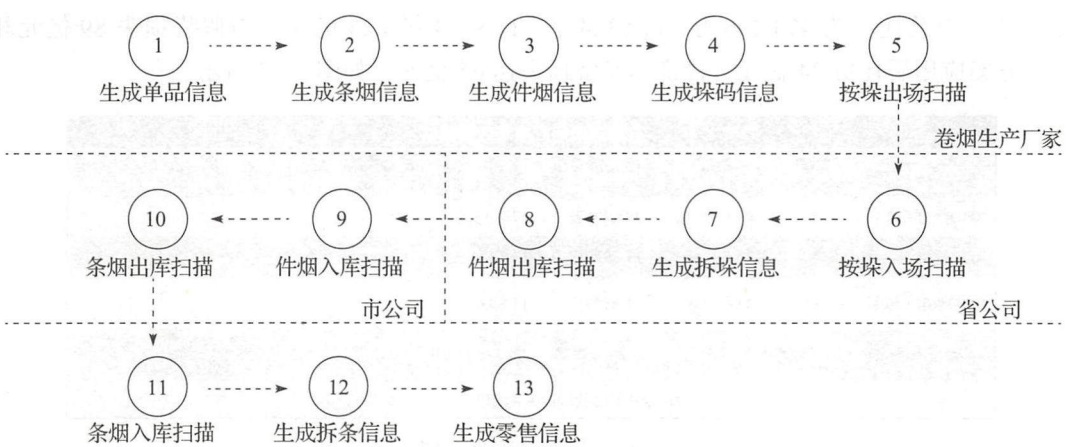


图 7-12　烟草防伪溯源系统数据流程图

6. 防伪判断及上报机制

鼓励消费者注册真实身份，通过验证真伪来积分奖励；与各烟草生产厂家协调，从打假资金中划专款奖励消费者，发动“人民打假”，降低打假成本。

仅供非商业用途或交流学习使用

扫到真标识码时的展示如图 7-13（来源网络上的示意图）所示。

扫到假标识码时的展示如图 7-14（来源网络上的示意图）所示。

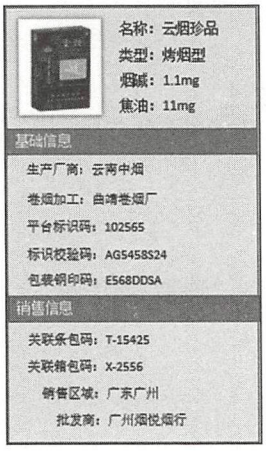


图 7-13 扫到真标识码时的展示图

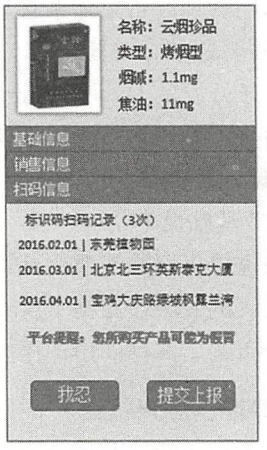


图 7-14 扫到假标识码时的展示图

7. 厂商投资收益预估

以云烟为例：云烟 2016 年出货，394608(1 个月货量)*12=473.296 万箱，1 箱 250 标准条，1 个标准条 10 个标准包（盒），总盒数：118.324 亿。云烟现售 50 种烟，只有 1/5 是 10 元以下的，估算均价 15 元，总价值 1774.86 亿元。按假货市场规模是正货的 15% 估算，假货总价值 266.229 亿，去除 177 亿成本，共计损失 89 亿元。

一标准箱 RFID 费用为 1 元，以标准条 RFID 费用为 1 元，一标准包 OFID 费用为 0.1 元计算，则总耗费为 473.296 万 +11.8324 亿 +11.8324 亿≈ 24 亿元。与假货损失 89 亿元相比，方案应用后耗费 24 亿元直接挽回经济损失达 65 亿元，如图 7-15 所示。

方案应用后厂商年效益预估						
项目	单价（元/张）	总量（张）	付出/损失（元）	现付/损失（元）	节省/挽回损失（元）	备注
RFID费用（箱）	1	473.296万	473.296万	473.296万	----	
RFID费用（条）	1	11.8324亿	11.8324亿	11.8324亿	----	
OFID费用（包）	0.1	118.324亿	11.8324亿	11.8324亿	----	
假货市场	----	------	266亿	177亿	89亿	方案应用把假货市场规模压缩到10%
为厂商节约或挽回损失总值预估：					>65亿	

图 7-15 厂商投资收益预估图

7.3.4 小结

对于烟草专卖局来说，希望实现全流通环节的跟踪和监控，打击违法造假经营，防止窜货发生；帮助政府通过识别“回收香烟”发现腐败线索；对卷烟销售情况进行统计分析，

仅供非商业用途或交流学习使用

有效控制企业生产节奏。

对于烟草企业来说，满足企业正向溯源的需求，随时掌握假冒烟草的信息；建立企业与消费者无障碍沟通平台，增加消费者黏度；有利于获取消费者信息，开展精准营销；提高企业和产品的知名度。

对于消费者来说，满足消费者反向溯源的需求，可以放心购买产品；有助于消费者快速了解企业和产品信息；利用智能手机查询真伪，简单快捷。

区块链方案的优势如下。

（1）数据真实可靠

基于区块链的共识机制，每个节点共同维护相同的数据库；RFID 电子标签不易被复制，数据安全性较高。

（2）责权清晰

基于区块链的不可篡改和可追溯特性，明确跟踪从生产到零售各个环节主体的操作，从“源头”遏制售假和窜货。

（3）信息全面归集

基于区块链的平权共建特性，构建联盟链生态圈，共建共享数据；产业链中各个环节都可以提供信息，且通过共识后上链保存，更方便信息的汇聚；RFID 电子标签不需要读取光束，可读取范围达几米远，便于信息采集。

（4）信息实时追踪

基于区块链的数据同步，信息在多方之间的流动将得到实时追踪和管理；对于大批量的制假行为能够清晰地识别出来。

7.4　区块链在海事稽查中的应用

7.4.1　简述

原先，海事局依托开具电子签证来放行船只，对过往船只收港口建设费等 7 项费用。现在，响应国家简政便民政策，减免了两项费用，电子签证系统也取消了。原本通过电子签证来控制船只出行，达到收取港口建设费等费用的目的，而现在没有电子签证，船只不缴费也能放行，导致海事局大量港口建设费也被偷逃了。所以海事局想建设稽查系统，来堵住这个缺口。

目前各地海事局数据相互间还没有共享，而船只收费的规则也比较复杂，比如只在始发港收费，中转港不收费，到保税区也不能收费等。这些原因导致稽查队伍拦截船只稽查收费时，没有权威有效的收费依据。

7.4.2　区块链解决的关键问题

1. 建设目标

1）构建跨区域、跨组织机构的海事稽查管理平台。

仅供非商业用途或交流学习使用

2）满足多方业务需求：登记、收费、检验、稽查。

3）构建业务标准，构建信任模型，提高业务效率。

2. 项目重点

1）原电子签证系统的取消，使得在船只放行时缺少凭证；需要构建新型凭证，用于标识船只和管理系统的关系。

2）各地海事局数据不共享，实现信息传递的方法有以下几种。

- 纸质凭证：可信度低、效率低。
- 中心化信息共享平台：安全性低，信任成本高。
- 区块链技术共享平台：信任成本低、安全性高。

3. 业务实施建议

1）各地海事局共同推行稽查管理信息共享方案，与各地海事局达成共识，将船只稽查管理有关的信息通过规范的方式共享发布到数据共享平台上，包括：航线信息、航班信息、场站信息、舱单信息、箱号信息、缴费信息、检验信息、稽查信息等。参与方包括：海事局、场站方、船只方、检疫检验局等。信息共享的业务动作包括：航班航线信息发布、场站装箱信息发布、装船信息发布、缴费信息发布、检验信息发布、稽查信息发布等。操作平台包括：PC 端功能、移动手持设备（APP）功能等。

2）以电子签证唯一性标识航班信息和箱号信息（一航班一码、一箱号一码），为箱号信息生成标识码并张贴，通过移动手持设备（APP）扫码识别获取共享的信息。为航班信息生成标识码，用于稽查时读取所有业务有关的共享信息。

7.4.3 方案描述

1. 业务组网

为参与方集成区块链节点，提供功能，如图 7-16 所示。

2. 逻辑架构

1）采用区块链技术为跨境业务的多个参与方提供平权互信的分布式业务数据操作共享平台，包括舱单、箱号、缴费功能。为各参与方发放身份识别的安全设备，在接入区块链网络后进行标准业务操作。

2）在业务逻辑层通过智能合约接口对接各参与方自有的业务系统，完成业务操作，将数据同时写入区块链平台和业务方自有业务系统。

3）平台提供稽查功能，辅助业务跟踪。

逻辑架构图如图 7-17 所示。

3. 系统功能架构

系统功能架构如图 7-18 所示。

仅供非商业用途或交流学习使用

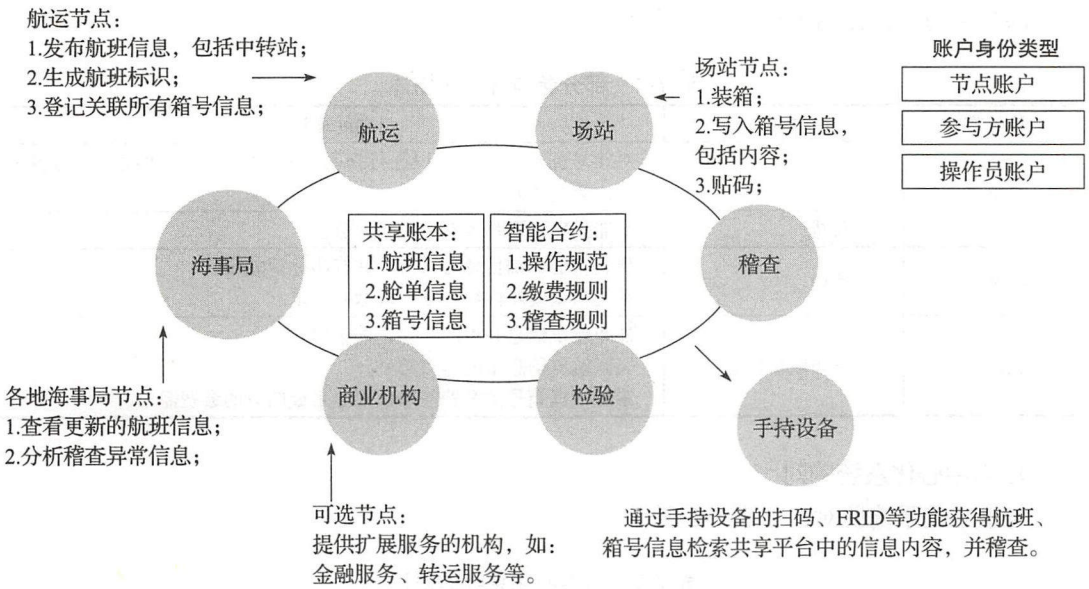


图 7-16　业务组网图

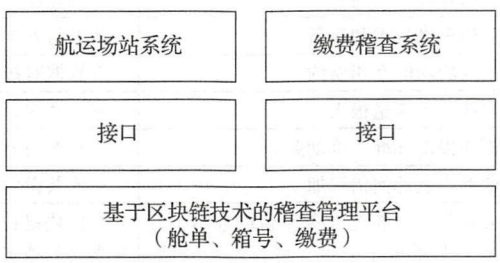


图 7-17　逻辑架构图

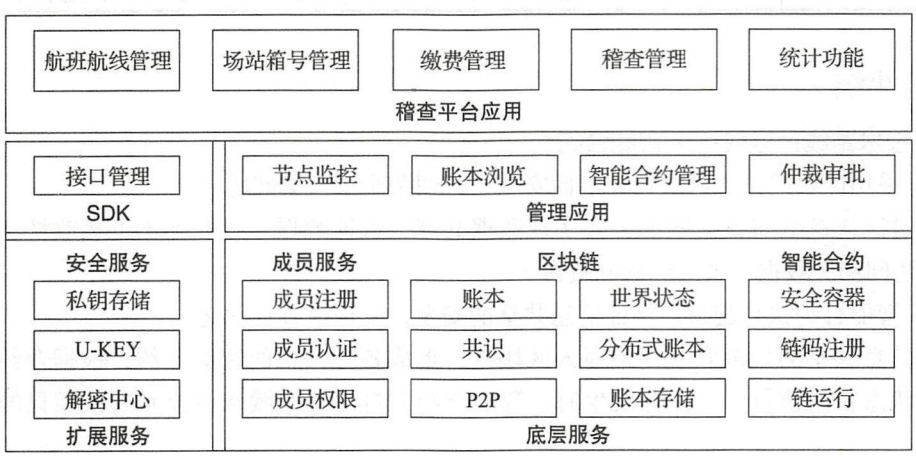


图 7-18　系统功能架构图

仅供非商业用途或交流学习使用

部分参与方操作描述如表 7-1 所示。

表 7-1 部分参与方操作描述

参与方	功能	功能描述
海事局	扫码稽查	对箱号标识、航班标识进行扫码（FRID）后，对比读取的内容和实际内容是否匹配
	收费	对船只项目进行收费，并写入链上
场站	装箱	将装箱的详细信息写入链，并打印标识贴箱 系统根据装箱内容自动生成缴费需求
航运	发布航班信息	发布航班信息并生成标识，携带出航 关联航班信息上的所有箱号 系统根据箱号的缴费需求，汇总生成船只的缴费需求

4. 与中心化系统的对比

与中心化系统的对比如表 7-2 所示。

表 7-2 与中心化系统的对比

对比项	中心化系统	区块链平台
建设方式	核心机构建设，其他参与	共同建设一套系统，一起使用
运维方式	核心机构提供技术和运营	各参与方维护自有节点
对接自身原系统	两个系统的功能互相制约	以数据对接的方式并存，无制约
对接他方系统	逐个对接，工作量很大	同一套规则，无须沟通和协调
业务实现方式	核心机构提出标准，推动业务	所有参与方制定标准，一起参与
业务实现难度	实现业务难度几何倍增加	达成共识后，不再有难度
可行性	以中心化的方式构建业务平台的成本十分高，业务参与难度十分大，可行性十分低。随着业务的发展，中心化系统面临适用性下降，调整战略的难度也十分巨大	在区块链技术的支撑下，构建多方共识平权参与的业务模式，具有低成本、业务参与难度低、高适用性等特点。随着业务参与方的增多，组网难度并不会增加，可行性高

7.4.4 小结

1. 技术路线：区块链 + 智能合约

1）采用区块链技术解决数据层面安全、透明和全面归集问题。

通过区块链分布式记账的方式实现数据共享，保证数据一致性和不可篡改性，各个节点均可访问相关数据，节约流程和资源。

2）智能合约执行交易，保证信息共享的安全，实现稽查自动化。

将信息共享的规范和规则编写入区块链，形成各方共识的智能合约，根据合约的规则和运行状态自动执行。在业务发生时，智能合约会自动根据缴费情况判断该船只的稽查状态，如：

预设好哪些类型的箱号需要缴费，在缴费端口会自动提醒收费；预设好哪些类型的箱

仅供非商业用途或交流学习使用

号在始发港、中转港的缴费，在对应港口自动提醒收费；根据航班情况、缴费情况提醒哪些航班仍需稽查（提醒未校验、未缴费）。

2. 后期运营可能性

1）提供业务有关的服务，如：管理服务、支付服务、改配服务。

2）提供延伸的增量服务，如：商品溯源服务、订舱服务、转运服务、金融服务、决策服务（类似数据魔方）。

7.5　区块链在教育领域的应用

7.5.1　简述

2016 年 10 月，工信部颁布《中国区块链技术和应用发展白皮书》，指出“区块链系统的透明化、数据不可篡改等特征，完全适用于学生征信管理、升学就业、学术、资质证明、产学合作等方面，对教育就业的健康发展具有重要的价值”。

7.5.2　区块链解决的关键问题

1）加强知识产权保护，搭建教育信任体系。

2）优化教育业务流程，实现高效、低廉的教育资源交易。

3）利用去中心化特性构建去中心化教育系统。

4）分布式存储与记录可信学习数据，实现校企之间高效对接。

5）开发教育智能合约，构建网络资源及平台运行新模式。

7.5.3　方案描述

区块链互教育领域相关应用如图 7-19 所示。

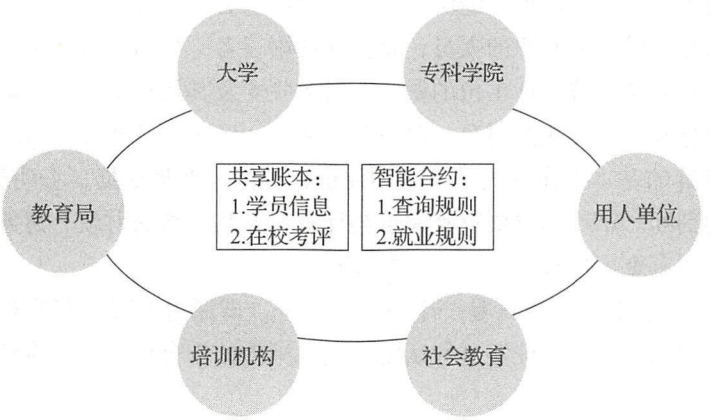


图 7-19　区块链相关应用

仅供非商业用途或交流学习使用

我们可以通过区块链的多方参与，共同维护同一个账本的形式，尽可能地引入多方信息。参与方越多，共同维护的数据越多，越容易给学生带来更多的数据信任背书，同时也能更加全面地描述学生本人各方面素质和能力。

区块链自身去中心化的特征，分布式的网络天然克服了中心化系统的各种弊端。同时，去中心化的特性避免了人为作恶的可能，避免了一部分人员通过人际关系修改数据的可能性。

通过区块链对学生学习数据全程上链，不仅仅贯穿学生生活始终，也可以为后续学生信息进入征信系统或者其他系统提供不可篡改的真实信息。

在整个学生数据上链的模式上，还要打通学生这一端。比如说我们可以联动家长、学校、老师这一端，配合更多的教学资源，通过学生积分奖励机制来提高学生对上链数据的积极性。让学生意识到引入区块链不只是一个数据上链，有利于学校、用人单位，而是对于其本人也有更多因为优秀而带来的额外奖励。

7.5.4 小结

区块链是一种在世界范围内学习领域有显著应用的技术，涉及个人、机构、团体、国家和国际层面，在各种情况下都是相关的，比如针对学校、学院、大学、网络公开课、CPD、企业、学徒制和知识库方面。与旧的层次结构不同，技术成为焦点，信任转移向技术，而不是机构。这实际上是一种去中介技术。

7.6 区块链在审计领域的应用

7.6.1 背景

审计是一项具有独立性的经济监督活动，独立性是审计区别于其他经济监督的特征。审计的基本职能不仅是监督，更是经济监督，是以第三者身份所实施的监督。审计的主体是从事审计工作的专职机构或专职的人员，是独立的第三者，如国家审计机关、会计师事务所及其人员。审计的对象是被审计单位的财政、财务收支及其他经济活动，这就是说审计对象不仅包括会计信息及其所反映的财政、财务收支活动，还包括其他经济信息及其所反映的其他经济活动。审计的基本工作方式是审查和评价，也即是搜集证据，查明事实，对照标准，做出好坏优劣的判断。审计不仅要审查评价会计资料及其反映的财政、财务收支的真实性和合法性，而且还要审查评价有关经济活动的效益性。

审计的通用定义如下："审计是一个系统化过程，即通过客观地获取和评价有关经济活动与经济事项认定的证据，以证实这些认定与既定标准的符合程度，并将结果传达给有关使用者。"

审计包括国家审计、社会审计和内部审计。审计工作流程如下。

仅供非商业用途或交流学习使用

（1）审计准备阶段

接受委托，确定审计项目；组成审计小组；了解和调查被审计单位；确定审计工作重点，制定审计工作计划方案；编制审计通知书、被审计单位承诺书；下发审计通知书等。

（2）审计实施阶段

对被审计单位提供的有关会计资料（包括电子文档）实施财务审计程序；整理、汇总审计过程中发现问题和有关情况，确定需要进一步核实的问题；实施相关的追加审计程序等。

（3）审计报告阶段

审计小组归集工作底稿，并编制审计报告初稿；审计小组按审核后的要求，对审计报告进行修改；审计小组汇报审计报告的征求意见稿，并对审计报告的征求意见稿进行定稿；就审计报告的征求意见稿征求被审计单位意见；出具正式审计报告等。最后审计小组将审计档案归档。

审计在现实中起到的作用是显著的，首先起到制约的作用，制止违规违纪现象，保护国家财产和企业的利益，有利于经济的稳步前行。其次能够促进相关部门改善经营管理，发现影响财务成果的各种因素，并提出解决问题的措施，进一步挖潜降耗，促进相关部门或企业的良性发展。

7.6.2　区块链解决的关键问题

首先在会计电算化系统中，传统的账簿、相关的文字记录被磁盘和磁带取代，加上从原始数据进入计算机，到财务报表的输出，会计处理集中由计算机按程序自动完成，传统的审计线索在这里消失。而审计线索的改变，导致在电算化系统中可人为篡改数据而不留痕迹，如电算化系统数据来源、公式定义、编制结果、打印格式均采用机内文件的形式，若有人篡改公式、编制失真的财务报表，然后再将篡改的公式等予以复原，则很难判定报表数据的正确与真实性。从而使得传统审计的追踪审查已不适用，审计入手点更多的是靠判断和经验。这样会给审计工作带来很大的麻烦，审计人员需要学会判断哪些数据被人为篡改了。其次审计人员从收集各种数据，然后到根据各种数据，做出统计、判断等，需要消耗大量的时间，工作量非常大。最后，审计工作的安全性和成本也很成问题，一般将审计数据存储于一台审计中心服务器上，不仅存在负载高、运行速度慢等问题，而且容易受到攻击。而成本方面是指人员成本、硬件维护成本、数据库软件升级成本等。

区块链审计系统的建立和完善给审计领域带来明显的好处。

1）区块链可以改进审计中的数据记录方式。现行的联网审计中，虽然有审计预警机制，但仍然需要审计人员对于异常记录进行人工判断与手工处理，区块链则可以通过各个节点是否对区块和其内的交易信息进行验证并认可，网络节点是否受到攻击，各节点的账本是否完整等信息，对异常记录进行自动处理，使实时审计成为可能。

2）审计人员可以直接访问查询区块链上的有效信息，判断处理是否合理并进行修正。区块链中采用时间戳来记录各项交易与操作，可以实现历史溯源与追踪，极大地提高审计质

仅供非商业用途或交流学习使用

量与效率。

3）区块链可以改变审计数据的存储方式。传统审计中，都将数据存储于一台审计中心服务器上，不仅存在负载高、运行速度慢等问题，而且容易受到攻击。而区块链审计系统则是典型的分布式存储，每个节点均有相同备份，不仅可以节省服务器的高额成本与维护费用，更重要的是保障了数据的完整性。

4）可以采用半公开私有链形式做到实时审计。区块链分为公有链、半公开私有链、完全私有链 3 种形式，而考虑到审计行业的特点，适宜采取半公开私有链形式。对于被审计单位，企业内部分商业机密信息不予以公开，而在其集团内部的预选节点来决定区块的生成，外部供应商等可以参与交易但不过问记账过程。对外则提供第三方查询节点，通过开放的 API 来进行查询。这样，可以在保证企业内部的私密性的情况下，使外部审计人员实施实时审计查询。

7.6.3 方案描述

利用区块链分布式存储结构实现数据资料实时获取、处理和存储。随着信息技术的快速发展，被审计单位的信息化程度不断提高，区块链去中心化的分布式存储结构能够在不同网络环境下对被审计单位的数据资料进行实时获取。实时审计系统通过标准的数据传输接口，与被审计单位的财务处理系统互连，企业经营活动中发生的每一笔交易数据被实时地传输到区块链网络上，上传的交易数据经过全网节点的批准，存储到区块链上。去中心化的分布式存储结构使得审计人员实时掌握被审计单位经营活动的一举一动。利用区块链技术的自治性和信息不可篡改的特性，可防止被审计单位私自篡改会计资料，保证被审计单位的数据资料真实可靠。

利用区块链应用平台，可在线实时进行经营风险评估、审计异常预警、实时审计和在线进行监督和审查。区块链应用平台包括：财务处理系统应用服务层、区块链技术资源层、实时审计应用服务层及实时审计访问层，如图 7-20 所示。

7.6.4 小结

区块链在审计工作中能大大降低调整多方复杂的不同信息时的错误率。此外，区块链中的财务记录一旦生成就不能改变，即使是会计系统的管理员也无法更改。因为每笔交易都会被记录和验证，财务记录的真实性得以保证。这样有利于审计工作的有效开展，也使得审计人员不用像以前那么辛苦的查证核对每个项目清单，有效地降低了审计人员的劳动强度。当然我们也应当认清现在区块链在应用方面面临的问题，首先，区块链的不可删除的特性使得链条会越来越长，占有磁盘存储量会越来越大，对每台作为节点的计算机，可能会造成负荷问题；另外一点就是区块链系统存在外部达到“51%”算力攻击，就可以改变或者操作账簿。这些问题都依赖于后续区块链技术的不断发展和完善来解决。现在区块链的大部分应用还处在小众阶段，或者说是实验阶段，但是区块链的发展前景是广阔的，当技术水平达到一定的程度，我们有理由相信，区块链在未来某个时刻会颠覆审计领域。

仅供非商业用途或交流学习使用

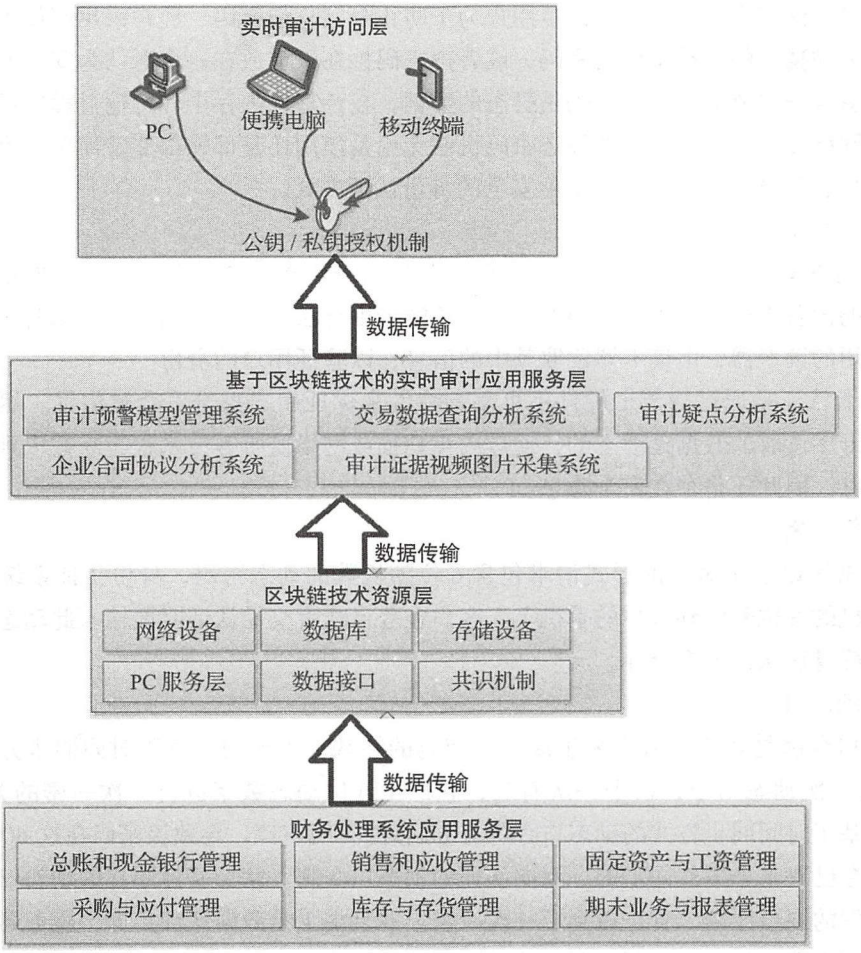


图 7-20　区块链审计应用

7.7　区块链身份认证

7.7.1　背景

身份认证也称为“身份验证”或“身份鉴别”，是指在计算机及计算机网络系统中确认操作者身份的过程，从而确定该用户是否具有对某种资源的访问和使用权限，进而使计算机和网络系统的访问策略能够可靠、有效地执行，防止攻击者假冒合法用户获得资源的访问权限，保证系统和数据的安全，以及授权访问者的合法利益。当前身份认证方法主要有如下几种。

（1）静态密码

用户的密码是由用户自己设定的密码。在网络登录时输入正确的密码，计算机就认为

仅供非商业用途或交流学习使用

操作者就是合法用户。实际上，许多用户为了防止自己忘记密码，经常采用诸如生日、电话号码等容易被猜测的字符串作为密码，或者把密码抄在纸上放在一个自认为安全的地方，这样很容易造成密码泄漏。如果密码是静态的数据，在计算机内存中和传输过程中可能会被木马程序或网络黑客所截获。虽然静态密码机制无论是使用还是部署都非常简单，但从安全性上讲，用户名 + 密码的方式一种是不安全的身份认证方式。

（2）智能卡

一种内置集成电路的芯片，芯片中存有与用户身份相关的数据。智能卡由专门的厂商通过专门的设备生产，是不可复制的硬件。智能卡由合法用户随身携带，登录时必须将智能卡插入专用的读卡器，由读卡器读取其中的信息，以验证用户的身份。

智能卡认证是通过智能卡硬件不可复制特性来保证用户身份不会被仿冒。然而由于每次从智能卡中读取的数据是静态的，通过内存扫描或网络监听等技术很容易截取到用户的身份验证信息，因此还是存在安全隐患。

（3）短信密码

短信密码以手机短信的形式请求包含 6 位随机数的动态密码，身份认证系统以短信形式发送随机的 6 位密码到客户的手机上。客户在登录或者交易认证时候输入此动态密码，从而确保系统身份认证的安全性。

（4）动态口令

动态口令牌是客户手持用来生成动态密码的终端，主流的是基于时间同步方式的，每 60 秒变换一次动态口令，口令一次有效，它产生 6 位动态数字进行一次一密的方式认证。但是由于基于时间同步方式的动态口令牌有 60 秒的时间窗口，导致该密码在这 60 秒内存在风险。现在已有基于事件同步的、双向认证的动态口令牌。基于事件同步的动态口令是以用户动作触发的同步原则，真正做到了一次一密。并且由于是双向认证，即：服务器验证客户端，并且客户端也需要验证服务器，从而达到了彻底杜绝木马网站入侵的目的。

身份认证存在的问题如下：

（1）信息分散

各个系统或部门都有自己的一套身份认证体系，采用的身份认证方法各不相同，安全等级也参差不齐。对用户来讲，首先面临重复录入的问题。用户使用每个系统之前都要注册账户，并输入个人信息，这严重增加系统使用的复杂度。而且各个系统录入信息的维度和要求不相同，个人偏好的一些账号昵称存在重复的现象，这样会导致个人信息维护困难。有的人隔一段时间不登录系统会忘记自己的 ID 或密码，导致个人无法登录，进而丢失自己在该系统上维护的重要信息。其次增加信息泄露风险。多个部门和系统中都存储个人信息，增加了个人信息泄露点。各个系统身份认证策略不同，安全方案也不同。难免存在安全方案薄弱的系统，容易被黑客攻击造成信息泄露。而且，有一些不法商家会使用用户的个人信息进行非法牟利，造成信息泄露并且难以追究其责任。由于系统和部门之间信息不能复用，更加不能形成完整的信息链，无法形成统一个人行为的证明，势必形成信息的孤岛。

仅供非商业用途或交流学习使用

（2）无监督审核机制

身份认证系统大多采用集中服务器进行存储和认证，控制权完全掌握在机构与部门手中。系统中的部门或一些不法商家为了自身的利益可以采用技术手段删除、修改个人身份信息或者个人行为信息，给个人信息安全带来很大隐患。中心认证体系中的信息容易受到攻击和篡改。并且个人信息没有长久记录审查机制，给伪造个人信息带来可能。中心认证体系也会存在节点不稳定问题，或因网络拥堵造成认证不成功等情况。一旦中心认证节点失能，就会到导致整个系统失效，极有可能造成个人信息丢失。

（3）保密性差

目前认证体系采用的安全加密手段各不相同，没有统一的认证标准。很多系统或商家为了在最短时间内使利益的最大化，并不会在身份认证体系上投入很多，没有通过安全认证，存在很多安全漏洞。很多商家的身份认证体系采用简单的数据库存储、明文通信等，攻击者非常容易获取和伪造个人信息。

（4）身份验证难

身份认证体系存储在多个系统和商家中，没有统一信息链，为身份验证带来困难。当涉及多个部门或者商家联合验证或取证的时候会遇到很多困难。而且各部门之间的身份认证信息很难做到格式统一。当遇到“如何证明你就是你”的问题时，涉及多个系统之间复杂的联合验证过程，将严重阻碍整个业务功能。

7.7.2　区块链解决的关键问题

用区块链作为身份认证体系能够保证身份信息的安全性、真实性、不可篡改性以及可追溯性，充分契合了需要身份认证的使用场景。

一个以区块链技术构建的身份认证体系能够统一管理身份信息，为多个部门系统提供身份认证服务。如图 7-21 所示是通过对个人信息进行认证，建立个人信息体系的一个示意模型。

其关键特性如下：

（1）可信的身份认证和开放

采用区块链存储加密后的个人基本信息，并为个人信息建立多个层次。根据不同应用场景或使用部门的需求，授予不同级别的访问权限。

区块链中能完整记录个人信息变更过程，且这个变更过程将配以时间戳被永久记录在账本中。这就为信息的真实性、完整性以及可追溯性提供了完美的解决方案。

使用智能合约编制个人身份认证的多种应用场景接口，使得区块链身份认证体系能够根据技术变迁及场景变化，及时提供高效、安全的认证服务。

（2）多部门系统的高效协同

各部门或系统可以直接采用区块链身份认证系统进行身份认证服务。由于区块链身份系统的可靠性、唯一性和安全性，可以保证系统部门的服务系统安全、高效地工作。各部门或系统不需要单独搭建身份认证体系，大大节约了构建和维护成本。

仅供非商业用途或交流学习使用

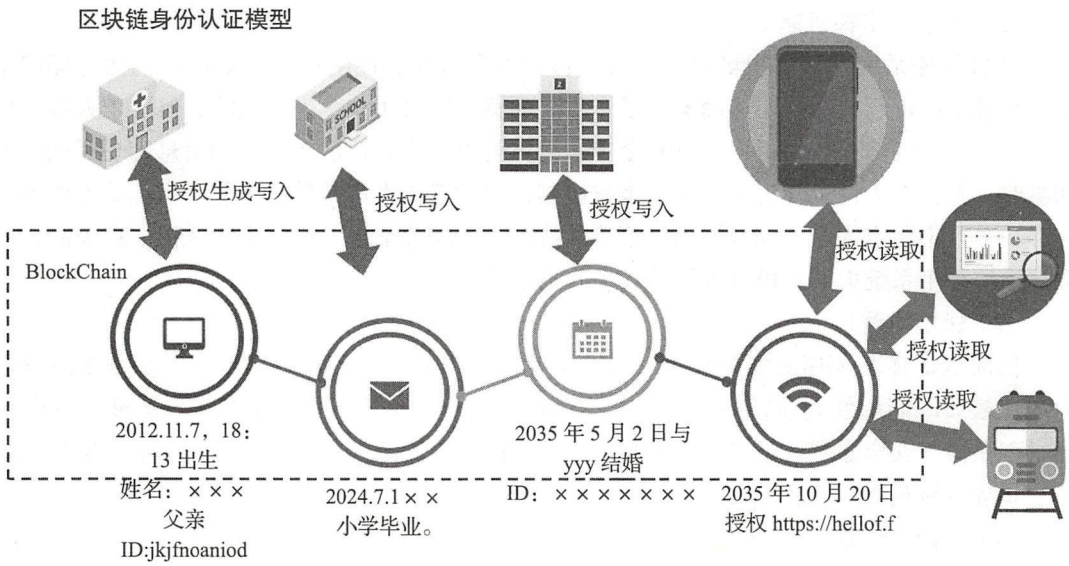


图 7-21　区块链身份认证模型图

各部门可以对个人进行信息授权、自理认证等。比如学校可以直接在链上记录毕业证信息，政府部门记录个人产权信息，银行记录个人财务信息等。

系统和部门之间可以充分做到身份信息的共享与验证，提高工作效率。因为各部门的认证信息可以在区块链身份认证体系中通过授权直接获取，而且认证信息都有相关部门的签名认证，保证了信息的权威性和安全性，节省了身份认证的时间和成本。

（3）完善的授权写入和读取验证机制

区块链的智能合约能够根据各种身份认证的场景完成授权、信息写入读取以及验证等多种接口，并可实现灵活扩展，具备开放、安全、灵活等特点。

7.7.3　方案整体架构

区块链身份认证系统采用区块链的链式结构存储个人身份信息以及变更的全过程。区块链具备不可篡改、使用签名技术进行交易等特点。这些特点能够最大限度地保证区块链所存储的个人信息的真实可靠。从个人账号创立就采用生物技术采样，获取个人唯一表示信息，生成个人账号，并创立最初始信息。之后伴随着个人的成长，各机关部门以及社会团体通过数字签名的方式对个人信息进行认证和添加信息。这些个人信息对个人的当前状态组成一个完整的信息链，给现实中的个人完成背书。

由于区块链采用在交易记录里面加时间戳的方法将交易信息固定在某个区块中，该区块中同时记录的所有信息都可以作为认证消息或者交易真实性的佐证。这样一个特性能最大限度地杜绝身份及认证信息的伪造和篡改。区块链身份认证体系除了在底层采用最基本的区块链技术进行存储外，智能合约方面还需要进行大量开发，以支持使用身份认证过程中的各种

仅供非商业用途或交流学习使用

场景。主要有账号的建立，账号信息授权写入，账号信息授权读取，账号联合认证及审查等。

区块链 ID 实现方案如图 7-22 所示。

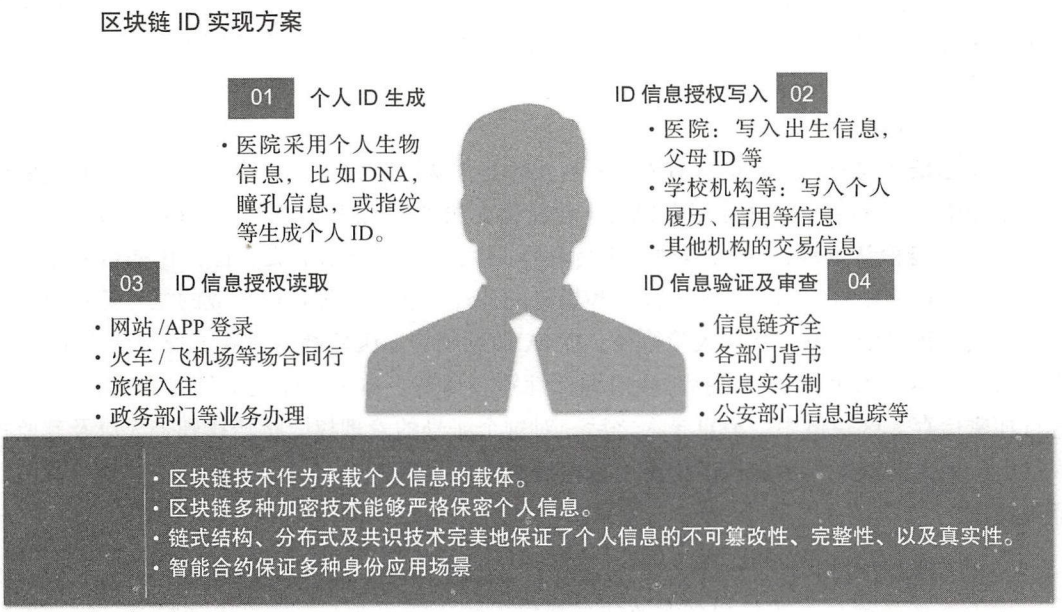


图 7-22　区块链 ID 实现方案

方案主要实现以下 4 个功能。

1. 个人 ID 生成

现代社会中，婴儿一般都在正规医院出生。医院会给每一个新生儿办理出生证明。出生证明上会记录该新生儿出生日期和时间，以及身长、体重、父母姓名等信息，并盖有医院的公章。出生证明将作为一个合法社会个体的起源的唯一证明，在后续的社会生活中扮演极其重要的作用。比如，凭借出生证明办理身份证，凭借出生证明入托等。出生证明作为一个社会人的起源证明，使用范围广泛，意义重大。但是，当前出生证明还是以纸质媒介作为载体，会带来诸多弊端：

（1）易伪造

出身证明虽有多种防伪手段，但是所有使用部门不可能都有验伪的设备和手段，这还是给出生证明带来了伪造的空间，由此进一步增加了社会犯罪的隐患。比如贩卖儿童现象，犯罪人员就可以通过伪造证件的手段将贩卖儿童的身份洗白，减少了犯罪的技术难度。另外，即使验证部门具备验伪的设备和技术，由于多了验伪这个环节，必然会给业务办理增加额外环节，影响业务办理的效率。

（2）易损毁丢失

个人保管出身证明证书很容易出现保管不善、损毁和丢失等现象，给个人的业务办理

带来麻烦，也给业务部门带来额外的工作负担。

（3）信息孤岛

政府学校组织等各部门要进行出生信息核验的时候只能通过人工查看、手工信息录入的手段实现业务的办理，不仅业务办理效率低下，而且各部门信息重复记录，无法实现业务办理自动化。

（4）无法记录个人生物特征

由于是纸质媒介，记录信息有限，所有出生证明无法记录更多与个人生理特征相关的信息，比如指纹、DNA 等。由于最关键的生物特性无法进行记录和比对，会有身份冒用隐患。

区块链身份认证体系采用多种生物特性作为区块链上公私生成凭证，比如可采用指纹信息生成区块链账号。当然，随着技术和场景的变迁，多种生理特征生成的账号信息可以用来管理同一组身份信息，比如人脸特征、DNA 信息码等。医院部门建立了身份认证账号后，将个人最基本的出生信息，包括出生时间、性别、身高、体重，以及父母双方的 ID 等信息进行加密后存入区块链中。消息录入之后，对这个账号的管理权限将转移到个人以及其监护人手中。个人对该账号中记录的信息可以按照级别逐层解密透露给第三方，以实现身份的认证等操作。

2. ID 信息授权写入

每个个体的成长需要得到各有关部门和组织的认证，个人信息也必须随着时间的推移而不断丰富。区块链身份系统必须包含个人身份信息的写入功能。个人身份信息包括学历、技能证书、驾驶执照、银行账号等。这些身份信息由相关部门采用相关区块链上该部门的账号，通过交易的形式进行加密写入，每笔信息的写入都使用数字技术进行区块链交易，保证了该认证信息的真实性、可靠性。并且写入的信息也通过个人账号的公钥进行加密，只有掌握该账号私钥的个人才有权查看和授权分级使用。每个提供身份信息写入的部门必须通过认证才能接入区块链网络，这就最大限度地保证了认证信息的准确和真实性。

3. ID 信息授权读取

个人账号及身份的读取、认证是区块链身份认证体系最大的使用场景。身份认证的范围可以延伸到方方面面。有需要匿名的网页浏览、投票、举报等；有需要实名的网上购物、购票、同行等。所有身份信息必须在拥有账号实际控制人的授权下才能使用，每次身份信息的使用都作为一次交易记录在区块链中。区块链中的身份信息进行分层加密和授权，有昵称、别名等匿名信息，有学历、证书、个人基本信息等真实信息。分层加密管理的个人信息可以在个人遇到不同的授权认证的场景下自主选择所能展示的不同级别的个人信息。在最大限度保护个人隐私的前提下，充分灵活地为各种身份认证场景提供解决方案。

4. ID 信息验证及审查

区块链身份认证体系由于其独特的链式结构以及信息的不可修改性，给需要多个部门联合认证的事务办理带来最大便利性和真实性，给公安以及司法等部门的审查取证也带来了

最充分和可靠的证据链。我们经常听见一些“如何证明你就是你”以及“你妈就是你妈”的问题，这些问题初听觉得可笑，但仔细思考其背后的本质是由缺乏完整的身份认证体系，以及缺乏多个部门为某个人进行联合背书所引起的。区块链身份体系从个体一出生就开始进行身份信息的管理，并在身份信息体系中加入了该父母对应身份信息。每次进行身份信息授权写入的时候都需要得到个人的授权，以及相关部门的数字签字才能完成。所有这些写入的信息无形中通过技术手段就得到了多个部门的联合认证。当公安部门和司法部门对某个嫌疑人进行取证的时候，由于区块链上能够将该人的所有信息都以交易信息的形式记录在案，并且区块链又能完整地保证记录信息的时间性和不可篡改性，这就从技术上保证了取证证据链的连贯性和真实性。

7.7.4　小结

区块链身份体系采用了分布式存储的区块链技术，将个人及单位的信息进行集中管理。充分利用了区块链的技术和集中管理的优势，在最大程度保护个人信息的基础上保证了信息的真实可靠，为联合认证、信息共享、完整信息链提供了可能性。

7.8　区块链在数据流通中的应用

7.8.1　背景

数据流通是指在数据提供方和数据需求方之间按照一定流通规则进行的、以数据为对象的行为。这种情况下，数据脱离了原有使用场景，变更了使用目的。随着数据的资源价值逐渐得到认可，以及大数据产业链结构日益完整，我国对数据流通的需求也日益迫切。无论是共享还是交易，数据流通都使数据从数据产生端转移至数据应用端，优化了资源配置，正在成为释放数据价值的重要环节。

根据数据使用的不同需求，流通的数据通常包括原始数据和加工处理后的衍生数据，涉及数据应用程序编程接口（application programming interface，API）、数据报告、原始数据分组、技术算法、数据应用等不同类型的数据资源商品。结合不同的数据使用需求，数据流通服务提供者通常采用不同的流通模式，如仅为提供方和需求方建立数据流通或服务关系的“中介型”服务；自主采集数据并对外销售的“采产销型”服务；对数据进一步加工处理，产生有价值的衍生数据或应用，并对外提供的“加工服务型”服务。在这些流通模式下，数据提供方和数据需求方通常不是同一实体。

然而，我国的数据流通，特别是交易产业数据流通，仍然面临着严重的问题，如数据隐私保护问题极其突出，数据权属问题需要得到准确界定，数据流通各环节标准缺乏统一共识，非法数据交易猖獗等。此外，随着《中华人民共和国网络安全法》的正式实施，非法贩卖数据正式入刑。在配套法规和标准不尽完善的情况下，许多企业为了规避风险，纷纷暂缓

或缩小了数据流通产业规模。数据流通产业面临近年来最大的挑战，亟须通过新的技术应用提供新的保障。

近年来兴起的区块链技术，从技术角度对上述问题提供了一种解决思路。下面主要介绍区块链技术解决数据流通中的几个关键应用场景，并梳理利用区块链实现数据流通的整体架构。

7.8.2 区块链解决的关键问题

通过区块链技术，可以在授权存证、数据溯源等数据流通的关键问题上进行改善，同时实现智能合约等新交易手段。

1. 利用区块链改造授权存证环节

长期以来，由于数据流通方、加工方、使用方的分离，数据二次交易没有手段稽核及管控，无法实时校验授权真实性，数据交易授权在技术层面并没有过多进展，往往采用如图 7-23 所示的传统模式。在这种模式下，用户通过数据提供方或数据交易机构等中介机构进行一对一、单独的授权。

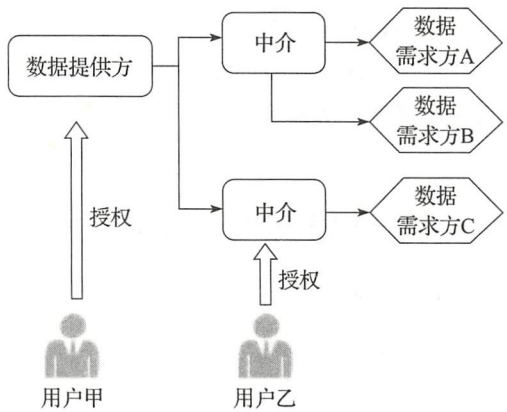


图 7-23 数据交易用户授权存证的传统模式

在传统模式下，授权存证可以被任意篡改，不具备公信力。由于需要相应责任认定条款，每个应用和数据源公司都需要单独签署协议。此外，查询授权记录需要单独开发接口，而这一点往往被忽略。由于授权和业务流程绑定，用户加入和退出都较为困难。

区块链技术的发展使得这一领域产生了新的突破。在区块链模式下，完整的授权鉴权流程如图 7-24 所示。用户签署电子协议，授予数据提供方相应权限。数据提供方首先通过应用系统本地存储凭证，进而将授权信息上传至授权信息链。应用系统执行链上代码，发起链上查询，并记录授权信息到区块。当数据需求方提交数据需求时，在链上发起鉴权交易，确认用户是否授权。接下来，链上验证节点返回授权信息，如确已授权，则返回相应数据。

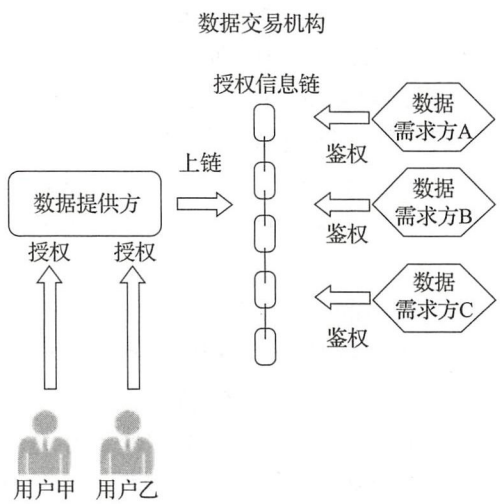


图 7-24 数据交易用户授权存证的区块链模式

区块链模式避免了传统模式的缺陷。任何节点可以记录授权信息，且不可更改。多方可以实时共享授权记录，查询效率较高。此外，授权与业务解耦，可随时加入和退出。

基于区块链推出政务服务数据共享交换平台，利用区块链技术的平权、共建的特点，依据共建共享的原则实现全面数据归集；通过区块链发布可信任的证照信息，实现原始发布部门数字签名的不可篡改和数据可信；基于区块链技术的非对称加密特点，对每条信息进行单独加密（每条公民的信息有单独的解密私钥），防止信息泄露；加速推行“互联网 + 政务服务”，打造政务服务“一张网”，创新实现政务服务数据在市级范围内跨区域的信息归集、快速检索和结果应用，未来拓展到全省的政务服务数据共享交换平台，打造可信的省市级政府政务信息共享开放平台，保障政府各职能部门之间的数据共享开放安全。

2. 利用区块链进行数据溯源

区块链技术的开放性、自治性、去中心化的特点非常适合数据交易（流通）溯源。目前，很多专家持有这样一种观点：数据 + 区块链 = 数据资产。区块链的价值就体现在对数据构造出了某种程度的“唯一性”。将带有唯一标识的数据附于区块链上进行交易，很自然地解决了数据难以溯源的问题。

区块链网络中多个参与计算的节点共同参与数据计算和记录，并且互相验证其信息有效性，既可进行信息防伪，又提供了可追溯路径。通过信息上链，各个区块的交易信息即构成完整的交易明细清单，不可篡改地记录了每笔交易的来龙去脉。当用户对某个区块的值有疑问时，可以准确方便地回溯交易记录，进而对历史交易记录进行判别。

当前，基于区块链的供应链管控与溯源技术正在取得快速进展。例如，可以结合区块链、比特币相关技术和多重签名技术设计供应链管控和溯源方案。将供应链内部实体分为“人物实体”“产品实体”和“权限实体”，将分层钱包技术用于实体密钥的分配。构建基于分层钱包技术的树形结构编码体系，确立基于区块链交易的去中心化权限管控机制和物权转移信息记录与验证机制，从而提出利用区块链实现供应链管控与溯源的新思路。

例如，食品安全一直是困扰人们的一大难题。以粮食为例，真正的五常大米年产量不过 100 多万吨，而市场上销售的五常大米的数量却超过 1500 万吨，老百姓很难确认吃到的是否是真正的五常大米。2017 年 4 月，智链（ChainNova）公司携手我国北方著名农场，运用基于 Hyperledger Fabric 1.0 的区块链技术和公钥基础设施（public key infrastructure，PKI）体系认证用户身份，打造了农业区块链应用。此应用对提交请求进行数字签名并上传至区块链，从而保证数据真实且不可篡改。同时，结合物联网、大数据等技术，打通链上链下的全流程数据通道，实现了 1296 万亩黑土地大米的追踪溯源与品质保障，将真正放心的高品质大米送到老百姓手中。

3. 基于智能合约实现数据交易

智能合约由计算机科学家、加密大师尼克·萨博于 1994 年首次提出，是一个能够自动执行合约条款的计算机程序，即一个预先编好的程序代码，其对从外部获得的数据信息进行识别并判断。当满足程序设定的条件时，即触发系统自动执行相应的合约条款，以此完成交易和智能资产的转移。然而，此概念提出后，因缺乏相应平台执行合约而处于被埋没的

状态。

区块链技术的出现使智能合约重新被关注和研究。区块链技术中的分布式账本结构贯穿了业务层（如资产）、应用层（如智能合约）、中间件层（如分布式交易共识）和底层技术层（底层网络）。智能合约能够在应用层上进行存储、验证和执行，因此智能合约成为区块链技术应用的重要特征。

以数据交易为例，赋予资产一些代码并在区块链上运行，使其成为全网共享资源，再通过外部数据触发智能合约执行，以决定网络中数据资产的流通、分配或转移。智能合约的标的物并不限于数据，可以是汽车、房子等物质产权，也可以是股权、票据、数字货币等非物质产权。

智能合约不仅由代码定义，还由代码强制执行，因此智能合约完全自动且无法干预，合约双方无须彼此信任。这恰恰符合数据交易的需要。数据交易机构可以通过建立规则，并用代码表述形式代替合同，实现链上支付功能，提高自动化交易水平。

7.8.3 方案整体架构

利用区块链技术实现数据流通，可以从网络交换层、共识机制层、数据存储层、智能合约层和数据流通层 5 个层次进行梳理。这 5 个层次如图 7-25 所示。

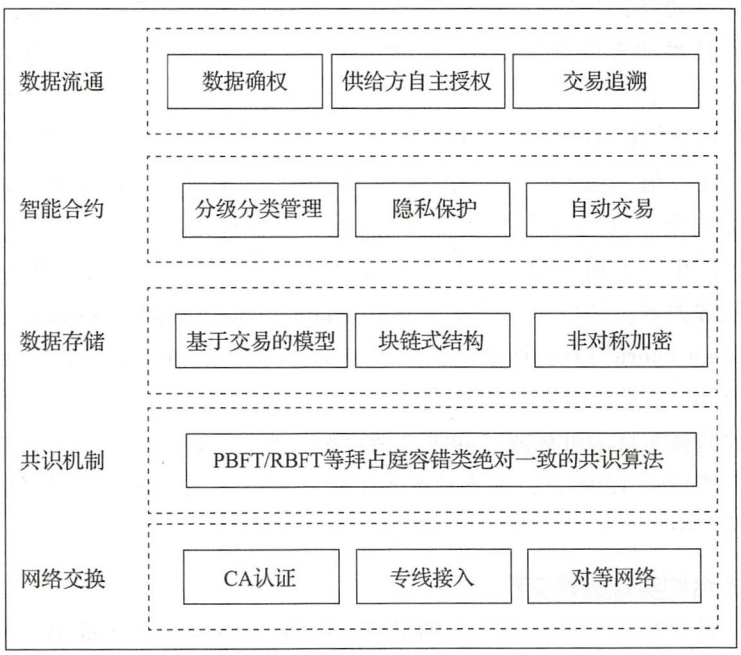


图 7-25 区块链实现数据流通的 5 个层次

在网络交换层，根据我国国家标准《信息系统安全等级保护基本要求》，非许可链（又

称公有链）的技术架构在物理访问控制、网络安全保障、服务性能要求、系统可靠运行等方面并不适应国家的相关规定，因此，非许可链的架构并不适应信息系统的等级保护规定，只能采用许可链（又称联盟链或私有链）的方式进行部署，需要通过专线接入、构建虚拟私有网（virtual private network，VPN）等方式保证通信的安全可靠。在身份认证方面，使用行业类或者区域类的电子商务认证授权机构（CA 认证）进行身份鉴权和接入控制，例如，做金融大数据交易的企业在进行接入时，需要通过中国人民银行下属的中国金融认证中心（China Financial Certification Authority，CFCA）的认证；在上海做大数据交易的机构进行接入时，需要通过上海市政府授权建立的上海市数字证书认证中心有限公司（SHECA）的认证等。整个网络是一种对等网络（peer-to peer），系统中多个节点的失效、退出和加入，甚至是恶意节点的存在，都不会影响整个系统运行的稳定性和安全性。

在共识机制层，区块链共识机制可以按照共识过程分为两类：第一类是概率一致的共识，工程学上最终确认，如工作量证明（Proof of Work，PoW）机制、权益证明（Proof of Stake，PoS）机制等；第二类是绝对一致之后再共识，共识即确认，如拜占庭容错（BFT）以及基于相关算法的变种（实用拜占庭容错（PBFT）算法等）。如前所述，在许可链的范畴中，尽可能采用绝对一致的共识算法。

在数据存储层，每笔数据交易都要持续跟踪，采用不同于普通商业银行的复式记账法模型——基于交易的模型，即针对未花费交易输出（unspent transaction output，UTXO）模型，更适合数据交易的管控和溯源。并且，通过块链式数据结构实现环环相扣的历史交易信息，通过非对称密码学技术实现公私钥的加解密，有助于数据交易之前的数据确权和数据上链等操作。

在智能合约层，依靠智能合约在隔离沙箱中的独立执行和相互校对能力，可以通过代码编写精确表达、实现数据分级分类管理的权限控制，有助于实现多个政府委办局之间的层级映射，也有助于跨层级、跨部门、跨区域、跨平台、跨行业的多机构协作联动。在隐私保护方面，主要依托智能合约独立运行的沙箱环境，除了数据授权方和利益相关方，无人能够接触到相关数据，并且严格按照智能合约设置的数据查看权限进行数据访问，这从一定程度上保证了数据的隐私性。通过智能合约的自动执行衍生出数据的自动交易，有助于解决数据定价难、精准计费难、交易撮合难等诸多问题。

在数据流通层，依托网络交换层、共识机制层、数据存储层、智能合约层等相关机制，相较于传统的数据流通平台，数据提供方更容易对自己的数据进行加密后传输，方便实现数据的确权管理。数据流通平台面对加密后的数据，更容易自证清白，避免流通数据的泛滥复制。依托区块链系统的溯源功能，数据所有方可以跟踪数据的流通现状，并且数据的每次使用都需要数据所有方的授权验证，实现数据提供方对拥有数据的自主管控，数据需求方可以追溯数据的源头，确保数据分析的真实性，提升数据分析的精准性和有效性。

值得指出的是，为增强数据流通的可靠性和安全性，区块链在研发设计中可以更加关注以下几点：一是选择正确的共识算法，依据数据流通对时效性的要求，综合考虑共识算

法；二是选择合规的个人信息保护算法，发挥区块链的技术优势，严格进行个人信息保护；三是选择适合的区块链部署模式，依据数据流通的特点和安全性要求，以区块链为主，确定较为适宜的区块链部署模式；四是可以将加密算法与数据的分级分类机制有机结合，对不同级别、不同敏感度的数据采用不同成本的加密算法，从而整体提升数据流通效率。

7.8.4 小结

区块链通过建立一组公共账本，由网络中所有用户共同记录，保证信息的真实性与不可篡改性。这些特性使得区块链有望成为破解数据流通难题的有效工具。然而，区块链的性能瓶颈和延迟性问题也愈发明显。除了技术方面的问题，其搭建成本以及与现有系统的融合性都成为制约其未来发展的因素。未来区块链能否在数据流通中发挥更大的作用，还取决于很多其他因素。

7.9 区块链在供应链金融中的应用

7.9.1 背景

供应链金融业务发生于核心企业及其上下游供应商、经销商，保理公司、银行等资金方之间，涉及钱款、物流、票据等信息的交互和校验。传统的供应链金融主要依赖线下信息或资源的传递以及各方内部系统的处理来完成，不仅耗时长，而且由于各方之间缺乏信任的因素，物流、票据等数据的真实性很难得到保证，往往需要烦琐地审核过程，带来较高的风险以及人工、时间的成本。应用区块链技术于供应链金融行业，利用其不可篡改、分布式、可追溯、数据隐私保护等特性，建立供应链各参与方之间的信任，保障数据的安全与私密性，控制交易过程中的风险，提高效率，降低成本。

7.9.2 区块链解决的关键问题

1）供应链上的中小企业融资难，成本高。由于银行依赖的是核心企业的控货能力和调节销售能力，出于风控的考虑，银行仅愿意对核心企业有直接应付账款义务的上游供应商（限于一级供应商）提供保理业务，或对其下游经销商（一级经销商），提供预付款或者存货融资。

这就导致了有巨大融资需求的二级、三级等供应商 / 经销商的需求得不到满足，供应链金融的业务量受到限制，而中小企业得不到及时的融资易导致产品质量问题，会伤害整个供应链体系。

2）作为供应链金融的主要融资工具，现阶段的商业汇票、银行汇票使用场景受限，转让难度较大。商业汇票的使用受制于企业的信誉，银行汇票贴现的到账时间难以把控。同时，如果要把这些汇票进行转让，难度也不小。

因为在实际金融操作中，银行非常关注应收账款债权“转让通知”的法律效应，如果核心企业无法签回，银行不会愿意授信。据了解，银行对于签署这个债权“转让通知”的法律效应很谨慎，甚至要求核心企业的法人代表去银行当面签署，显然这种方式的操作难度极大。

3）供应链金融平台 / 核心企业系统难以自证清白，导致资金端风控成本居高不下。目前的供应链金融业务中，银行或其他资金端除了担心企业的还款能力和还款意愿以外，也很关心交易信息本身的真实性，而交易信息是由核心企业的 ERP 系统所记录的。

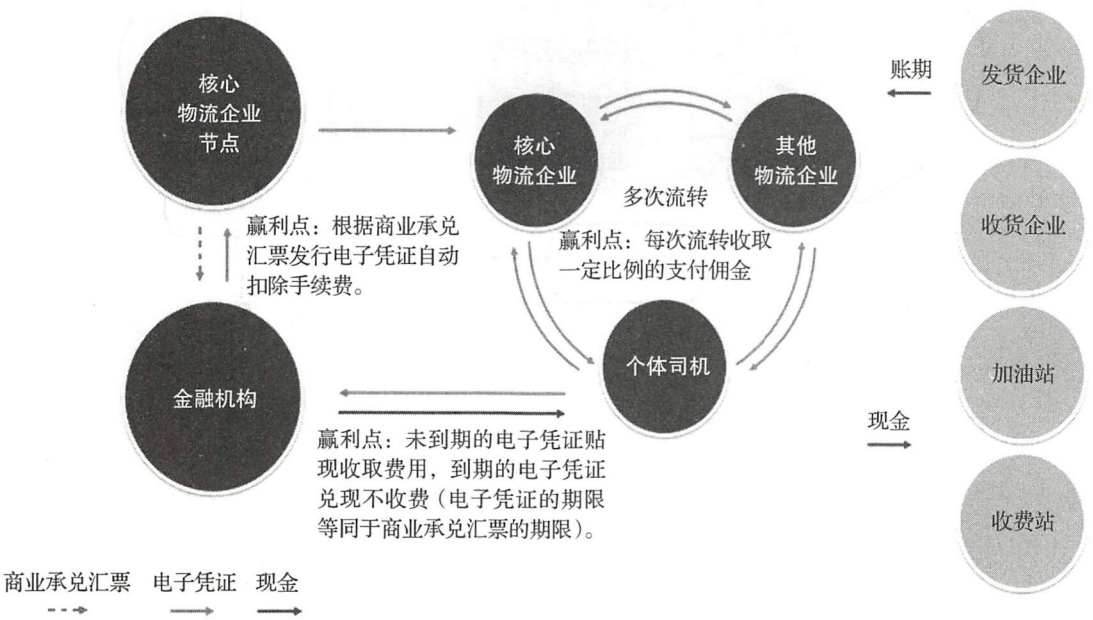
虽然 ERP 篡改难度大，但也非绝对可信，银行依然担心核心企业和供应商 / 经销商勾结修改信息，因而需要投入人力物力去验证交易的真伪，这就增加了额外的风控成本。

7.9.3　方案整体架构（以物流为例）

1）发行：金融机构与核心物流企业共同发行可信的电子凭证。

2）流通：在物流企业之间通过电子凭证结算往来的业务费用。

3）兑付：金融机构对所有电子凭证提供兑现（贴现）的保底。



特点：

1）现金托底：电子凭证的总额需匹配商业承兑汇票的总额。

2）灵活可信：电子凭证可以拆分成任何大小，且不可造假。

3）兑付约束：电子凭证需要和现实发票一起兑现（贴现）。

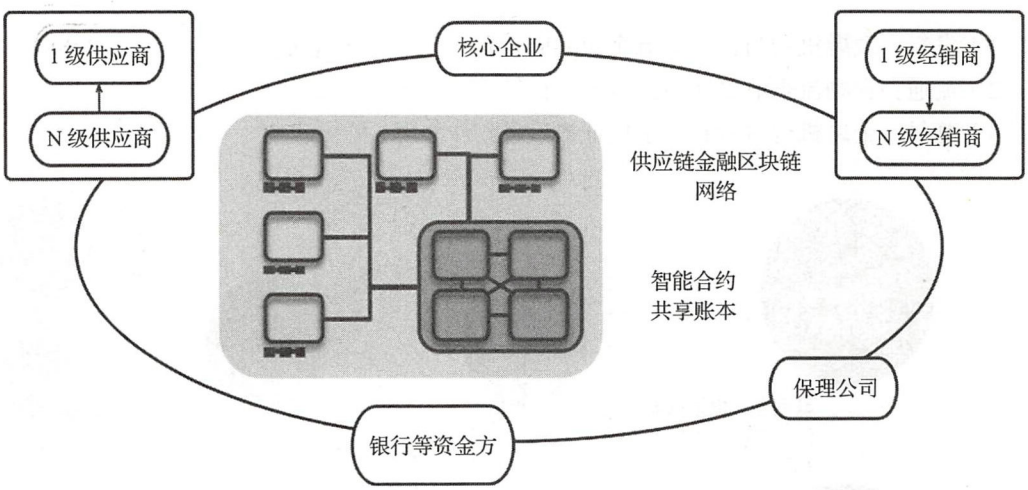
7.9.4 小结

1）对于资产方来说，应提高交易效率，降低交易成本，获得更多资金方的支持。解决核心企业及其上下游供应商、经销商在各自系统以及线下操作带来的数据安全性、可靠性问题，以及交易过程的效率和成本问题。解决资产方由于资源有限、数据难于被资金方认可的难题。

2）对于资金方来说，应控制交易风险，寻求更多优质资产方，解决保理公司、银行等资金方风控问题。

3）对于监管方来说，应增强交易监管解决交易数据的真实性与溯源问题；实现快速审查与调阅。

注：笔者曾经了解过明链科技供应链金融 SaaS 公有云平台方案，为供应链各个环节的企业提供存取证服务、标准及定制化的应收应付供应链业务服务。



第 8 章 Chapter 8

区块链未来展望

8.1 区块链与人工智能的关系[⊖]

人工智能（Artificial Intelligence，AI）是计算机科学的一个分支。它旨在揭示智能的实质，并生产出一种新的能以人类智能相似的方式做出反应的智能机器，该领域的研究包括机器人、语言识别、图像识别、自然语言处理和专家系统等。

区块链技术和人工智能中间的渊源可以概括为：区块链技术可以运用到人工智能中，大大提高人工智能的安全性和稳定性。很多人在人工智能的产物——机器人出现后，担心机器人是否会拥有自己的思维，从而毁灭人类，就像某电影里那样。但是区块链技术的发展速度非常快速，可运用的范围越来越广，如果真正将二者结合起来，将是一项更安全、不会威胁人类生存安全的伟大发明。

人工智能与区块链技术结合最大的意义在于，区块链技术能够为人工智能提供核心技能——贡献区块链技术的“链”功能，让人工智能的每一步“自主”运行和发展都得到记录和公开，从而促进人工智能功能的健全和安全、稳定性。区块链 + 人工智能将迎来一个新的时代，未来会进入智能互链时代，链在很大程度上就是区块链。在新的时代下，数据是新的生产资料，计算是新的生产力，而智能和区块链构成新的生产关系，区块链和人工智能是下一步社会生产关系变革的基础。

区块链与人工智能相结合，存在以下的机会。

机会 1：数据共享→更好的模型

简而言之：去中心化 / 共享式控制鼓励数据共享，这反过来带来了更好的模型，进而带

⊖ 本节摘自云头条，作者 Frent McConaghy，云头条团队翻译，亿欧编辑。

来了更高的利润和更低的成本等好处。

人工智能爱数据。数据越多，模型越完善。不过，数据常常是筒仓式（即孤岛式）的，在数据好比护城河的这个新环境下更是如此。

但是区块链鼓励在传统的孤岛之间共享数据，前提是有足够多前期好处。区块链的去中心化性质鼓励数据共享：如果没有哪个单一实体控制存储有数据的基础设施，共享面临的阻力比较小。

这种数据共享可能会出现在企业里面（比如在区域办事处之间）、生态系统里面（比如“联合”数据库），或者整个星球（比如共享式全球数据库，又叫公共区块链）。

1）在企业里面：来自不同区域办事处的数据使用区块链技术合并起来，因为它降低了企业审计自己数据的成本，还降低了与审计人员共享该数据的成本。若有了这些新数据，企业就能构建这种人工智能模型：比如能够比只能在区域办事处层面构建的之前模型更准确地预测客户流失率。这就相当于每个区域办事处的“数据集市”。

2）在生态系统里面：竞争对手（比如说银行或唱片公司）传统上根本不会共享其数据。但是不难理解，如果拥有来自几家银行的合并数据，一家银行可以构建更完善的模型，用于预防信用卡欺诈。或者对一条供应链上通过区块链共享数据的诸多企业来说，如何可以更准确地查明供应链中之后出现的故障的根源，针对来自供应链上游的数据使用人工智能。

3）整个星球（公共区块链数据库）：不妨考虑在不同的生态系统之间共享数据（比如能源使用方面的数据、汽车零部件供应链数据），或者每个个体参与全球规模的生态系统（比如 Web）。来自更多来源的更多数据可改进模型。比如说，一些工厂的能源使用激增可与出现在市面上的仿冒汽车零部件关联起来。总的来说，我们在聚合数据、清洁数据、重新包装并出售数据的公司身上看到这方面的迹象。

机会 2：数据共享→全新的模型

在一些情况下，来自孤岛的数据合并后，得到的不仅仅是更好的数据集，还能得到全新的模型，由此带来全新的模型。可以从该新模型获得新的洞察力，并获得新的商业应用。也就是说，可以做之前做不了的事情。

下面是识别钻石欺诈的一个例子。如果你是一家提供钻石保险的机构，很想构建一个可识别钻石是否冒牌货的分类器。全球有 4 家信誉卓著的钻石认证实验室（当然取决于你问的是谁）。如果你只能获得其中一家实验室的钻石数据，而对另外 3 家实验室的数据一无所知，你的分类器就很可能将通过那 3 家实验室鉴定的钻石标为欺诈性。你的误报率会让你的系统毫无用处。

不妨改而考虑如果区块链促成所有 4 家实验室共享数据，那会怎样。你将拥有所有合法的数据，你可以用来构建一个分类器。任何送来的钻石（比如 eBay 上在卖的钻石）都将通过该系统的审查，与这个所有数据的单类分类器进行比对。分类器可检测真正的冒牌货，避免误报，因而降低了欺诈率，从而惠及保险提供商和认证实验室。这可能只是被称为查询，即不需要人工智能。但是使用人工智能可进一步改进它，比如说根据颜色、克拉等方面

预测价格，然后使用“价格与预计价值多接近”作为主欺诈分类器的输入源。

这里是第 2 个例子。去中心化系统中一种合适的标记奖励方法可激励数据集加以标记（而之前无法加以标记），或者以一种经济高效的方式加以标记。有了新的标记，我们获得了新的数据集；我们训练新的数据集，以获得新的模型。

这里是第 3 个例子。标记奖励方法可以导致数据由物联网设备直接输入。设备控制数据，可以交换数据以获取资产，比如能源。这种新数据再次会带来新模型，这后两个例子要感谢迪米·德·扬赫（Dimi de Jonghe）。

囤积还是共享？两个相反的动机在这里形成对峙。一个是囤积数据——“数据是新的护城河”观点；另一个是共享数据，以获得更好 / 新的模型。要共享，势必要有一个足够明显的驱动因素压倒“护城河”带来的好处。技术驱动因素是可获得更好的模型或新的模型，但是这个驱动因素势必会带来业务好处，比如减少欺诈，节省钻石或供应链方面的保险费；可以在 Mechanical Turk 中捎带赚钱；数据 / 模型交换中心；对某个大玩家采取集体性行动，比如唱片公司可以集体起诉苹果公司的 iTune。好处不止这些，这需要创新的业务设计。

中心化还是去中心化？即使一些企业组织决定共享，它们还是可以在不需要区块链技术的情况下共享。比如说，它们可能仅仅要把数据聚合到 S3 实例中，并在它们本身之间公开 API。但是在一些情况下，去中心化带来了新的好处。先是名副其实地共享基础设施，那样共享联合体中的一家企业组织无法独自控制所有的“共享数据”（这在几年前是一块主要的绊脚石，那时唱片公司试图联合起来，建一个通用的注册中心）。另一个好处是，更容易把数据和模型变成资产，然后可以授权外面的人使用，以获得利润。感谢亚当·德雷克（Adam Drake）对囤积与共享这种对峙予以特别关注。

正如讨论的那样，数据和模型共享会出现在 3 个层面：企业里面（对跨国公司而言，其难度超乎想象）；生态系统或联合体里面；或者整个星球（这相当于成了一家公用事业公司）。不妨更深入地探讨全球规模的共享。

机会 3：新的全球规模的数据→新的全球规模的洞察力

全球规模的数据共享可能最值得关注。不妨进一步探讨这个方面。

IPDB 是全球规模的结构化数据，而不是零星的数据。万维网（WWW）好比是互联网上面的文件系统；IPDB 是其对应的数据库。我们没有更早看到这方面的更多工作，是由于从升级文件系统的角度来看，语义 Web 试图抵达成功的彼岸。但是通过“升级”文件系统来构建数据库却相当难！

那么，如果我们使用像 IPDB 这样的全球规模的共享数据库服务来共享数据，会是什么样子？我们有几个参考点。

第一个参考点是，已经有一个产值达到 10 亿美元的市场，许多公司精选并重新包装公共数据，让数据更容易使用，从用于天气或网络时间的简单 API，到股票和货币等金融数据，不一而足。设想一下：如果所有这些数据可通过单一数据库，以一种类似的结构化方式（即使它只是通过 API）来访问，会是怎样子。这相当于 1000 个彭博。

第二个参考点来自区块链，体现于这个概念：对外部数据进行 oraclizing 处理，通过区块链，让外部数据易于使用。但是我们可以对所有数据进行 oraclize。去中心化的彭博就是个开始。

总的来说，我们为众多数据库和数据源获得了全新的规模。因此，我们拥有全新的数据、全球规模的结构化数据。我们可以由此构建全新的模型，能够在输入和输出之间建立之前无法建立起来的关系。借助模型，我们可以从模型获得全新的洞察力。

还有机器人这个角度。我们一直假设：区块链 API 的主要使用者将是人类。但是如果是机器，又会怎样？现代 DNS 的开发者大卫·霍尔兹曼（David Holtzman）最近表示："IPDB 是人工智能的吊桶。"细细分析，那是由于 IPDB 支持和鼓励全球规模的数据共享，人工智能确实爱"吃数据"。

机会 4：针对数据和模型的审计跟踪记录，获得更可靠的预测

这种应用面对这种现实：如果你训练垃圾数据，就会得到垃圾模型。对测试数据来说也是如此。正可谓，垃圾进垃圾出。

垃圾可能来自恶意的家伙、可能篡改数据的错综复杂的故障。想一想大众公司尾气排放丑闻。垃圾还可能来自并非恶意的家伙、崩溃故障，比如来自有缺陷的物联网传感器、出故障的数据源，或者导致比特翻转的环境辐射（没有很好的纠错机制）。

你怎么知道输入 / 输出（X/Y）训练数据就没有缺陷？实时使用怎么样，针对实时输入数据运行模型？模型预测（yhat）怎么样？简而言之：进出模型的数据是什么情况？数据也想要信誉。

区块链技术可助一臂之力。方法如下：在构建模型以及在实际现场运行模型的过程的每一步，该数据的创建者只需要给该模型标以时间戳，并添加到区块链数据库，这包括对它进行数字签名处理，声称"目前我相信该数据 / 模型是好的。"进一步阐述如下。

构建模型方面的数据溯源：

1）传感器数据（包括物联网）方面的数据溯源。你信任你的物联网传感器告诉你的数据吗？

2）训练输入 / 输出（X/Y）数据方面的数据溯源。

3）构建自己的模型方面的数据溯源，如果你喜欢，可通过可信的执行基础设施或类似 TrueBit、复核计算的市场来进行。至少，要有证据表明使用构建模型的收敛曲线（比如 nmse vs.epoch）来构建模型。

4）模型本身方面的数据溯源。

测试 / 实际现场方面的数据溯源：

1）测试输入（X）数据方面的数据溯源。

2）模型模拟方面的数据溯源，如可信执行和 TrueBit 等。

3）测试输出（yhat）数据方面的数据溯源。

我们在构建模型和运用模型方面都获得了数据溯源。结果是获得了更可信的人工智能

训练数据和模型。

优势如下：

1）可以在所有层面发现数据供应链（从最广泛的意义上说）存在的泄露现象。比如说，你可以查明某传感器是否在“撒谎”。

2）能以一种可通过密码来验证的方式，了解数据和模型的情况。

3）可以发现数据供应链存在的泄露现象。那样一来，如果错误出现，就能极其清楚地知道错误为何出现、出现在哪里。可以把它看成是银行界的对账，不过核对的对象是人工智能模型。

4）数据得到了信誉，因为多双眼睛可检查同一数据源，甚至坚持自己的主张，表明它们认为数据有多有效。而与数据一样，模型也得到了信誉。

机会 5：训练数据和模型的共享式全球注册中心

人工智能界的一个特别的挑战是：数据集在哪里？传统上，它们分散在互联网上，不过有一些列表列出了主要的数据库。当然，许多数据集是专有的，就因为它们具有价值。还记得数据护城河吗？

但是，如果有一个全球数据库，易于管理另一个数据集或数据源（免费或收费），会怎样？这可能包括来自众多机器学习竞赛的广泛的 Kaggle 数据集、斯坦福大学的 ImageNetdataset 及其他无数的数据集。

这正是 IPDB 所做的。人们可以提交数据集，并使用别人的数据。数据本身会放在 IPFS 之类的去中心化文件系统；元数据（以及数据指针本身）将放在 IPDB 中。我们会获得一个人工智能数据集的全球共同体，这有助于实现开放数据社区的梦想。

我们不该止步于数据集，我们还可以加入用那些数据集构建的模型。获取和运行别人的模型，提交你自己的模型，这应该很容易。全球数据库会为此提供极大的便利。我们能获得由全球拥有的模型。

机会 6：数据和模型是 IP 资产→数据和模型交换中心

不妨深入叙述运用由训练数据和模型组成的“共享式全球注册中心”。数据和模型可能是共同体的一部分，但是它们也可以买卖。

数据和人工智能模型可以作为一种知识产权（IP）资产来使用，它们受版权法的保护。这意味着：

1）如果你构建了数据或模型，就能拥有版权。这是指你想不想用它来开展任何商业活动。

2）如果你拥有数据或模型的版权，那么就可以授权别人使用。比如说，你可以授权别人使用你的数据来构建自己的模型。或者，你可以授权别人把你的模型添加到其移动应用程序中。也可以层层授权：你授权别人使用，别人授权他人使用。当然，你也可以在获得授权后使用别人的数据或模型。

我认为你可以拥有人工智能模型的版权，并授权别人使用，这很棒。数据已经被认为是一个可能很巨大的市场，模型会亦步亦趋。

在区块链技术问世之前，就可以拥有数据和模型的版权，并授权别人使用。一段时间以来，相关法律为此提供了依据。但是区块链技术让它变得更好，原因如下：

1）就你拥有的版权而言，它提供了一个防止篡改的全球公共注册中心。你拥有的版权由你以数字方法或加密方法来签名。这个注册中心还包括数据和模型。

2）就你的授权交易而言，它再次提供了一个防止篡改的全球公共注册中心。这回，它不仅仅是数字签名，而是说，你甚至无法转让版权，除非拥有私钥。版权转让作为类似区块链的资产转让来进行。

有些专家很注重区块链方面的 IP，早在 2013 年就在开展 ascribe 方面的工作，帮助数字艺术家拿到应有的报酬。最初的方法在授权的规模和灵活性方面有问题。现在，这些问题已得到了解决。让这成为可能的技术如下：

1）Coala IP 是一种灵活的、对区块链友好的 IP 协议。

2）IPDB（以及 BigchainDB）是一种共享式公共区块链数据库，存储版权信息及其他元数据，规模堪比 Web。

3）IPFS 以及 Storj 或 FileCoin 之类的物理存储是一种去中心化文件系统，可以存储庞大的数据和模型 blob。

因此，我们得到了作为 IP 资产的数据和模型。

为了说明，我以 ascribe 为例，我拥有多年前构建的一个人工智能模型的版权。这个人工智能模型是决策树（CART），用于决定使用哪种模拟电路拓扑结构。这里，它是一种采用密码的防伪证明书（COA）。如果你想获得我的授权以便使用，只管发邮件给我。

一旦我们有了数据和模型这种资产，可以开始为那些资产建立交换中心。交换中心应该是中心化的，就像 DatastreamX 已经为数据建立的机制那样。但到目前为止，它们其实只能使用公开可用的数据源，因为许多公司认为共享带来的风险高于回报。

去中心化的数据和模型交换中心怎么样？如果在“交换中心”这种环境下实现数据共享去中心化，就会出现新的好处。由于去中心化，没有哪一个实体控制数据存储基础设施或表明谁拥有什么的账本，这样企业组织更容易协同工作或共享数据。不妨想一想用于深度网络（Deep Nets）的 OpenBazaar。

有了这样一种去中心化的交换中心，我们会看到真正开放的数据市场出现。这有望实现数据和人工智能人士长期以来怀有的梦想。

当然了，我们在那些交换中心上会有基于人工智能的算法交易：人工智能算法购买人工智能模型。人工智能交易算法甚至可能购买算法交易人工智能模型，然后更新自己。

机会 7：控制你数据和模型的上游

这承接前一种应用。

如果你注册使用社交网站，也就把它对你输入其系统的数据可以做什么、不可以做什么方面很具体的权限授予了社交网站，它有权使用你的个人数据。

当音乐家与唱片公司签约后，他们将非常具体的权限授予了这家唱片公司，比如编辑

音乐、发行音乐等。

对人工智能数据和人工智能模型来说可能一样。如果你构建的数据可用于构建模型，当你构建好模型，就可以预先指定许可证，限制上游的别人如何使用它们。

区块链技术为所有使用场合简化了这方面，从个人数据到音乐，从人工智能数据到人工智能模型，不一而足。在区块链数据库中，你把权限当成资产，比如说，读取权限或查看某一部分数据或模型的权限。作为权限拥有者，你可以把作为资产的这些权限转让给系统中的别人，就像转让比特币那样：创建转让交易，并用你的私钥来签名。

因此，你对于使用你的人工智能训练数据、人工智能模型及更多内容的上游有了极大的控制权。比如说，你可以重新混合这个数据，但不可以深度学习它。

这可能是 DeepMind 在医疗区块链项目中采用的战略的一部分。在数据挖掘中，医疗数据让它们面临监管风险和反托拉斯问题（在欧洲更是如此）。但是如果用户能改而真正拥有其医疗数据，并控制上游使用，那么 DeepMind 只要告诉消费者和监管者："客户实际拥有他们自己的数据，我们只能使用它。"劳伦斯·伦迪（Lawrence Lundy）提供了这个很棒的例子，他随后做了进一步的外推。

完全有这个可能：政府允许私人拥有（人类或 AGI）数据的唯一方式就是借助共享式数据基础设施，采用"网络中立"规则，就像 AT&T 和长长的原始线路那样。从这个意义上来说，日益自主的人工智能需要区块链及其他共享式数据基础设施得到政府的接受，因而从长远来看需要可持续发展。——劳伦斯·伦迪。

机会 8：人工智能 DAO—能积累财富，你无法关闭的人工智能

这个很出色。人工智能 DAO 是拥有自己，你无法关闭的人工智能。

到目前为止，我们谈论了作为去中心化数据库的区块链。但是我们也可以实现去中心化处理：基本上，存储状态机的状态。拥有这方面的一点基础设施让它更容易实现，而这就是以太坊（Ethereum）等"智能合约"技术的精髓。

我们之前也实现了流程去中心化，表现为计算机病毒。没有哪一个实体拥有或控制病毒，你无法关闭它们。但是它们是有限制的，它们基本上试图破坏你的计算机，就是那样。

但是如果你与这个流程有更丰富的交互，该流程本身可以独立积累财富，那会怎样？现在通过更好的 API，让这成为可能，比如智能合约语言，以及公共区块链之类的去中心化价值存储系统。

去中心化自治组织（DAO）这种流程体现了这些特点。代码可以拥有数据。

这给我们带来了人工智能。名为"强人工智能"（AGI）的人工智能子领域最密切相关。AGI 是指在环境下交互的自治代理。AGI 可以建模成反馈控制系统。这是好消息，因为控制系统有许多出色的特性。首先，它们有强大的运算基础，可以追溯到 20 世纪 50 年代诺伯特·维纳（Norbert Wiener）的"控制论"。它们捕获与外界的交互（驱动和感知），并适应（根据内部模型和外部传感器来更新状态）。控制系统使用广泛，它们控制简单的恒温器如何根据目标温度来调节；它们可以为你昂贵的耳机降噪；它们是另外众多设备的核心部件：从

微波炉到汽车制动器。

人工智能界最近更积极地拥抱控制系统。比如说，它们是 AlphaGo 的关键。AGI 代理本身就是控制系统。

人工智能 DAO 是一种类似 AGI 的控制系统，它在去中心化的处理和存储底层上运行。反馈回路自成一体，获得输入信息后，更新状态，驱动输出，并拥有能不断这么做的资源。

我们可获得人工智能 DAO，只要从人工智能（AGI 代理）入手，并让它去中心化。或者，我们可以从 DAO 入手，为它赋予人工智能决策功能。

人工智能得到了其缺失的一环：资源。DAO 得到了其缺失的一环：自主决策。正由于如此，人工智能 DAO 可能比人工智能本身或者 DAO 本身要庞大得多。潜在的影响是倍增的。

8.2 区块链与大数据

大数据需要应对海量化和快增长的存储，这要求底层硬件架构和文件系统在性价比上要大大高于传统技术，能够弹性扩张存储容量。Hadoop 的 HDFS 奠定了大数据存储技术的基础。另外，大数据对存储技术提出的另一个挑战是多种数据格式的适应能力，因此现在大数据底层的存储层不只是 HDFS，还有 HBase 和 Kudu 等存储架构。大数据的分析挖掘是数据密集型计算，需要巨大的分布式计算能力。节点管理、任务调度、容错和高可靠性是关键技术。Hadoop 的 MapReduce 是这种分布式计算技术的代表，通过添加服务器节点可线性扩展系统的总处理能力，在成本和可扩展性上都有巨大的优势。现在，除了批计算，大数据还包括了流计算、图计算、实时计算、交互查询等计算框架。大数据利用多台机器的计算资源，并将不能由单个机器处理的任务分配给多台计算机，通过处理不同的任务，每台计算机集成了多种计算资源，形成强大的数据处理能力。

我们知道区块链是比特币的底层技术架构，它在本质上是一种去中心化的分布式账本。区块链技术作为一种持续增长的、按序整理成区块的链式数据结构，通过网络中多个节点共同参与数据的计算和记录，并且互相验证其信息的有效性。从这一点来说，区块链技术也是一种特定的数据持久化技术。区块链的共识机制就是所有分布式节点之间怎么达成共识，通过算法来生成和更新数据，去认定一个记录的有效性，这既是认定的手段，也是防止篡改的手段。区块链每个节点计算机运行基本相同的任务，整个区块链通过重复的冗余计算，可以实现多个实体之间的互信。由于去中心化在安全、便捷方面的特性，很多业内人士看好其发展，认为它是对现有互联网技术的升级与补充。

我们可以看出，区块链和大数据采用的都是分布式存储和分布式计算，从系统架构上有相似之处。但是从本质上讲，两者的不同也是明显的。

1）结构化与非结构化：区块链是结构定义严谨的块，通过指针组成的链，典型的结构化数据；而大数据需要处理的更多的是非结构化数据。

2）独立与整合：区块链系统为保证安全性，信息是相对独立的；而大数据侧重的是信

息的整合分析。

3）直接与间接：区块链系统本身就是一个数据库；而大数据指的是对数据的深度分析和挖掘，是一种间接的数据。

4）数学与数据：区块链试图用数学说话，区块链主张“代码即法律”；而大数据试图用数据说话。

5）匿名与个性：区块链是匿名的（公开账本，匿名拥有者，相对于传统金融机构的公开账号，账本保密）；而大数据在意的是个性化。

表 8-1 从几个维度对比了大数据和区块链的异同。

表 8-1　大数据和区块链的异同

	大数据	区块链
共性 1	分布式架构：分布式存储和分布式计算	
共性 2	数据管理，广义数据库技术	
面对问题	海量数据，提高性能	关键数据，防篡改
计算模式	MapReduce 把一件事分给多个人去做	共识机制 多个人重复做一件事
存储结构	HDFS HBASE Kudu 等多种数据结构	单一的块链式数据结构
数据类型	非结构化	结构化
基础网络	服务器集群	P2P 网络
价值来源	数据是信息 从数据中提炼价值	数据是价值 实现价值的传递
CAP 理论	选择 AP	选择 CP

结合区块链与大数据的各自特点，可以产生巨大的化学反应。区块链让数据真正“放心”地流动起来。区块链以其可信任性、安全性和不可篡改性，让更多数据被解放出来。用一个典型案例来说明，即区块链是如何推进基因测序大数据产生的。区块链测序可以利用私钥限制访问权限，从而规避法律对个人获取基因数据的限制问题，并且利用分布式计算资源，低成本完成测序服务。区块链的安全性让测序成为工业化的解决方案，实现了全球规模的测序，从而推进数据的海量增长。区块链是一种不可篡改的、全历史的分布式数据库存储技术，巨大的区块链数据集合包含着每一笔交易的全部历史。随着区块链技术的应用迅速发展，数据规模会越来越大，不同业务场景区块链的数据融合会进一步扩大数据的规模和丰富性。

区块链以其可信任性、安全性和不可篡改性，让更多数据被解放出来，推进数据的海量增长。区块链的可追溯性使得数据从采集、交易、流通，以及计算分析的每一步记录都可以留存在区块链上，使得数据的质量获得前所未有的强信任背书，也保证了数据分析结果的正确性和数据挖掘的效果。区块链能够进一步规范数据的使用，精细化授权范围。脱敏后的数据交易流通，则有利于突破信息孤岛，建立数据横向流通机制，形成“社会化大数据”。

基于区块链的价值转移网络，逐步推动形成基于全球化的数据交易场景。区块链提供的是账本的完整性，数据统计分析的能力较弱。大数据则具备海量数据存储技术和灵活高效的分析技术，可极大提升区块链数据的价值和使用空间。

8.3 区块链即服务

8.3.1 概念

区块链作为一种分布式账本技术，在近两年的宣传和推广中逐步为业界接受。使用场景也从金融领域逐渐扩展到产业链、物流、医疗、农业等领域，前景无可限量。

区块链由点对点协议、分布式架构、密码学技术，以及共识技术等多种技术组成。技术覆盖面广、难度大，对于从业和维护人员技能要求高，维护门槛较高。很多有使用要求的行业和企业往往因为安装难度高，维护成本大而望而却步。在这样的背景下，各大云运营厂家结合各家的云平台开发出来区块链即服务（BlockChain as a Service，BCaaS）的云服务产品，让有使用区块链产品的企业或商家无须自己搭建和维护区块链系统，通过 BCaaS 就能享受区块链的云服务。

8.3.2 原理

BCaaS 是将各种区块链技术与云计算技术相结合的产物。各云运营商将各种区块链技术部署在自家的云上，对外提供区块链服务。可以说，BCaaS 是一种特殊的 SaaS（Software as a Service）。只不过区块链跟传统意义上的软件服务不同的地方是它是分布式架构的，不能简单地理解成某个虚机上的提供的软件服务。

BCaaS 可以简单地分为 5 层，前 4 层分别为物理机 / 云主机层、区块链层、智能合约层，以及 DApp（分布式应用）层。除了这 4 层之外还必须配置区块链浏览器以及系统管理。整个架构图如图 8-1 所示。

物理层是由物理服务器和网络设备组成的最基础的云平台基础设施。物理层是整个服务体系的基石和载体。云平台运营商也是今后物理层搭建和维护的主体。云平台运营商管理和运营硬件层，在此之上为客户提供计算和存储能力，而免去了客户的硬件和管理费用，这才有了云平台的市场。

云平台层是云平台运营商的云平台产品。比如亚马逊的 AWS、IBM 的 Bluemix、阿里的阿里云、腾讯的腾讯云等。各个云平台的特点以及使用方法各不相同。作为 BCaaS 的云平台，与传统的云平台服务有所不同。传统的云服务是集中是云服务。一个应用架设在某个云平台上很多情况下一台主机可能就可以实现。而在 BCaaS 中，由于区块链是分布式架构的所以构建区块链网络的时候必须同时启动多台云主机。而且根据云平台选用的区块链技术不同，使用的最小云主机数量也不相同。

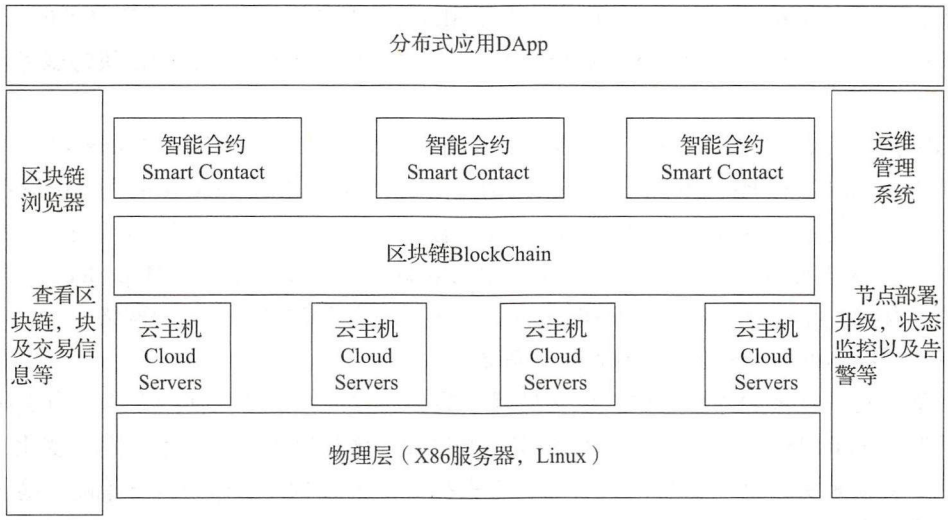


图 8-1　BCaaS 总体架构图

区块链层是各个云平台根据自身平台的特点选用的底层区块链技术。在选取区块链技术的时候，云平台要考虑场景和客户群体。从目前各家平台的技术选型来看，很多平台都选用社区影响力比较大的开源区块链技术。这是从技术的更新维护、客户的认同度、使用场景等多方面考虑的。很多厂家使用了 Linux 基金会的 Hyperledger Fabric，比如 IBM 的 Bluemix、Oracle Cloud、腾讯云 TBCaaS 等。也有集大成者，比如微软在 Azure 平台上就部署了 Hyperledger Fabric、Ethereum、R3 Single Network、Quorum 等。这种解决方案给用户更多选择，可根据业务灵活配置。

智能合约层是区块链和分布式应用的中间连接层。根据区块链的技术不同，智能合约的开发和使用形式各不相同。智能合约也是根据上层分布式应用的业务流程进行开发的，因此与业务强相关。

分布式应用 DApp 是区块链使用单位或者联盟成员的业务逻辑。根据不同的场景要进行定制化的开发。DApp 通过智能合约将与区块链进行状态交互。DApp 涉及多种行业领域，比如金融、产业链、物流、农业等。

区块链浏览器是对区块链信息进行查看的窗口，主要信息包括节点信息、区块信息，以及区块里包含的交易信息。

运维管理系统负责这个 BCaaS 的部署、升级、监控、告警的运行维护工作，为客户打造最方便快捷的区块链平台。

8.3.3　IBM 区块链服务

1. 简介

IBM 在全世界范围内是涉足区块链技术比较早的公司。Hyperledger Fabric 项目就是

IBM 进行开源并维护的，而且 IBM 的云平台 Bluemix 也是最早部署区块链服务的云平台。BCaaS（BlockChain as a Service）这个概念和称谓也是从 Bluemix 平台的区块链服务流行起来的。

IBM 的 BCaaS 项目被称为 IBM Blockchain Platform，是给予 Linux 基金会的 Hyperledger Fabric 和 Hyperledger Compser 构建而成。IBM Block Platform 通过 IBM Cloud，提供托管的全栈区块链即服务（BCaaS）产品。让成员能够开发、治理和运营网络，并且其性能和安全性能够满足监管行业要求。IBM Blockchain Platform 利用 Hyperledger Fabric，支持基于内容不可更改、共识信任和隐私保护原则建立的新型分布式商业网络。

（1）内容不可更改和数据一致性至关重要

一笔交易一旦被加入账本中，该交易就不能被商业网络中的任何一个参与方删除或变更。因为 Hyperledger Fabric 不会形成分支，所以加入区块链中的信息不会发生变化，除非用另一个交易来更新这些信息。只有当各方根据灵活的架构（也称为背书策略）签署了交易，交易才算达成。分布式账本技术必须支持企业联合开发共享的真实信息源，满足特定业务网络的要求。

（2）通过许可型背书而不是匿名方式建立信任

与无授权即可访问网络不同，Hyperledger Fabric 和 IBM Blockchain Platform 并非通过匿名方式信任。整个业务网络应该知道业务网络的参与者，从而在一个具名的业务网络中建立分布式信任。监管要求（包括 HIPAA 和 GDPR）往往会规定具名网络中的参与者和交易的提供者的某些信息。

（3）网络的隐私性

尽管参与者在网络上是具名的，但是他们在网络上交易时的隐私性和安全性是能够得到保证的。企业应对其交易数据和交易本身的安全性充满信心。当企业不想与整个网络共享某些信息时，Hyperledger Fabric 能够帮助企业通过专有渠道进行机密交流。托管 BCaaS 平台提供了最快速、最简单、最经济高效的方法，在各团队组织之间运行分布式的网络。随着区块链目的发展和成熟，从试验性的概念验证发展至分布式多方生产网络，IBM Blockchain Platform 能为区块链项目提供正确的工具和功能。

2. 架构

IBM Blockchain Platform 构建于关键开源工具之上，为企业提供开发、运行和治理企业解决方案的必备基础架构。IBM Blockchain Platform 的端到端架构如图 8-2 所示。该图结合了 400 多家客户的参与经验，基于企业级区块链网络提供商用环境。这是目前业内唯一的商用端到端平台，能支持企业以最快的时间启用分布式的区块链网络。目前，全世界众多企业都在生产环境中使用该架构。

3. 开发

认识交易业务网络价值的第一步是让开发人员能够将创新的商业理念变成现实。借助

仅供非商业用途或交流学习使用

IBM Blockchain Platform，开发人员能够利用通用工具和语言建立业务应用模型，构建和测试业务应用，并将业务应用部署到分布式的业务网络中。该平台能为开发人员提供以下帮助：

1）利用独特的建模语言，确保业务和技术之间保持密切协调，大幅度缩短区块链应用的开发周期。

2）利用 JavaScript 和 REST 等热门工具和语言，帮助现有程序员快速培养区块链技能。

3）利用开放的先进工具集，在首选的环境中灵活学习，并开发应用 IBM Blockchain Platform 构建于 Linux Foundation 的 Hyperledger 管控的两大开源项目之上：Hyperledger Fabric 和 Hyperledger Composer。在 Hyperledger Fabric 之上，由 Hyperledger Composer 提供使用常用编程语言和工具，快速构建区块链业务网络和设计业务应用程序原型的基础平台。

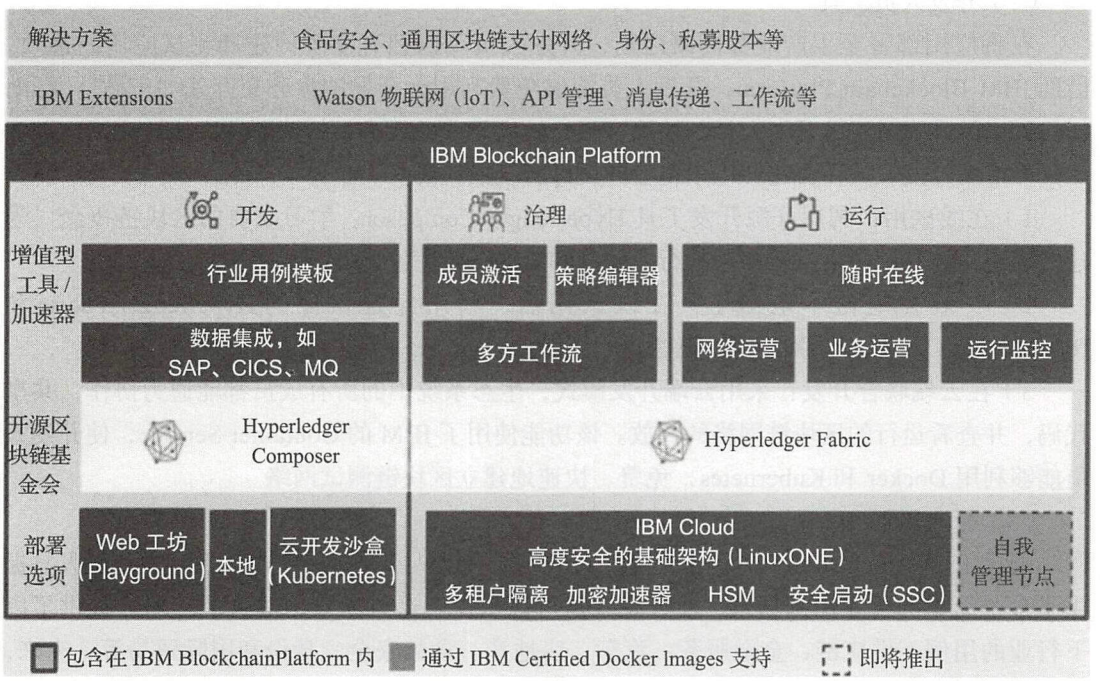


图 8-2　IBM BCaaS 架构

（1）Hyperledger Composer

Hyperledger Composer 为企业构建区块链应用提供了框架，这些应用将能反映业务网络的核心结构。这个框架能帮助开发人员：

1）建立业务网络模型。

2）通过自动生成的 REST API，呈现区块链数据和业务逻辑。

3）创建使用区块链数据的应用。

仅供非商业用途或交流学习使用

Hyperledger Composer 中包含强大的面向对象的业务：

1）域特定语言，旨在具体说明业务模式，包括资产的结构、参与者和交易。用户可以在 Hyperledger Composer 中使用业务域模型、API 生成、验证类型，用户界面生成等。

2）Hyperledger Composer 包含一系列代码库、数据模型和运行时、开发工具，以及基于 Web 的开发环境，旨在加速学习和采用。在应用开发流程中，所有这些功能不仅能降低风险，还能提高速度和效率。

3）Hyperledger Composer 围绕一系列工具包设计而成，这些工具是 Hyperledger Fabric 函数式单元的高层抽象，包括：基于 DSL 的 duchess 主模型，用于定义资产和关系；基于 JavaScript 定义的、面向业务逻辑的功能（即智能合约）；业务域模型中描述访问控制规则的表达式。IBM Blockchain Platform 基于 Hyperledger Composer 构建，让开发人员能够以安全且可重复的方式轻松构建应用，并将应用部署至实时的分布式的业务网络中。

（2）开发人员工具

在将应用部署至生产业务网络之前，开发人员可以拥有多个构建和测试应用的选项。借助 IBM Blockchain Platform，开发人员能够免费利用云开发沙盘和交互式“工坊”，帮助任何程序员成为区块链开发人员，快速、轻松地满足业务要求，加速开发区块链应用。以下工具可帮助开发人员在首选环境中将业务设计转变成代码。

1）在线试用：利用开源开发工具 Hyperledger Composer，学习重要的区块链概念，创建网络定义，并利用可重复使用的行业模型和智能合约库。

2）安装在笔记本电脑上：在线试用后，利用经过认证的 Hyperledger Fabric 和 Hyperledger Composer 的 Docker 镜像。

3）在云端联合开发：采用云端开发模式，生态系统中的所有成员都能通力协作，共享代码，并查看运行的区块链网络的回放。该功能使用了 IBM 的 Container Service，使开发人员能够利用 Docker 和 Kubernetes，免费、快速地建立区块链测试网络。

（3）行业用例

通过 IBM Blockchain Platform，开发人员无须一切从零开始构建。该平台为开发人员提供了多个简单的行业用例场景，使开发人员可以从探索这些场景用例入手。IBM 提供了以下行业的用例：供应链、金融服务、汽车、房地产、食品安全、身份和国际贸易等。未来，还将增加更多其他行业用例场景。

（4）轻松集成现有业务数据

IBM 认为，很多企业都想将区块链运营与他们的大量现有数据资产集成一体。为了在开发应用时更轻松地实现这种集成，IBM 提供了 API，帮助企业利用 Hyperledger Composer REST API 服务器，集成企业现有的 SoR（System of Record）。Hyperledger Composer 还利用 Node-Red，建立业务流模型；利用 LoopBack，辅助数据流路由。

IBM Blockchain Platform 支持多个开发选项，确保业务需求与技术能力保持一致。在了解智能合约领域易受攻击的编程语言后，用户无须单独集成多个协议和平台，即可确保业务

仅供非商业用途或交流学习使用

需求与技术功能保持一致。

4. 治理

构建分布式的业务网络时，最重要的功能是清晰、有效的治理定义、模型和工具。IBM Blockchain Platform 提供重要功能，确保构建的网络拥有定义清晰的模型，并且无须重启整个网络即可在需要时完成更新。构建好区块链网络后，如何在一组成员间启用和治理区块链网络，这需要大量协调工作、时间和精力。人们往往忽视和低估了妥善治理区块链网络的能力。通过妥善治理区块链网络，客户最终能确保网络符合规定，消除业务责任的不确定性和风险（体现在智能合约中），确保不同交易类型的隐私性和安全性（体现在渠道），并建立一个引入新成员的审批流程。IBM Blockchain Platform 提供的重要治理优势如下：

1）管理工具，让网络成员能够共同管理分布式的业务网络的治理规则和策略。

2）动态管理环境，从而随着网络的发展和新智能合约的产生，不断增加网络成员。

3）预构建工具，实现更快的入门、定制和激活。

IBM Blockchain Platform 提供了第一套集成工具，允许团队能够利用定制化的策略，跨队列执行网络变更管理。

（1）激活工具

随着新参与者和交易的创建，分布式的业务网络在不断变化。借助激活工具，网络成员能够轻松提高网络的规模，设置新的智能合约，并在更广泛的业务网络中构建渠道。

（2）策略编辑器

客户必须以灵活的方式支持区块链网络的核心组件，如共识、成员策略、智能合约和交易渠道。借助 IBM Blockchain Platform 中的策略编辑器，分布式的业务网络的（所有或部分）成员能够合作更新网络治理的策略。

（3）多方工作流工具

网络成员需要了解各方如何在网络上互动。IBM Blockchain Platform 提供了一个成员活动面板的工作流工具，展示集成式定制化通知，并在进行规则投票时，保证签名收集的安全。

（4）网络模型

与传统的业务网络一样，不同的参与者有不同的业务目的。IBM Blockchain Platform 能够将参与者设置成特定的角色，不同的业务目的，受不同的治理策略管制。IBM Blockchain Platform 上的分布式的业务网络成员能够扮演一个或多个角色，包括参与者、成员、用户，成员提供商或成员消费者。每位成员都能根据其业务需求运行多个对等节点，并参与不同的网络。可以配置通信“渠道”，确保只有特定的成员能够查看特定的数据。成员能够利用共识和排序的集群，提交并更新他们的账本副本。只有能够通过身份验证的应用，才能成为业务网络中交易的主用户界面。

5. 运行

处理任务关键型应用和交易数据的分布式的业务网络，需要构建于满足以下要求的平

仅供非商业用途或交流学习使用

台之上：支持安全且可扩展的“始终在线”的运行和更新。借助 IBM Blockchain Platform，成员能够利用生产就绪型、安全加固的服务，部署并运行分布式的网络。

（1）操作系统

IBM Blockchain Platform 的核心操作系统是 Hyperledger Fabric。该网络的后端操作环境运行于网络发起者选择的服务计划。对 Enterprise 和 Enterprise+ 选项感兴趣的使用者可以选择高度安全的 LinuxONE 基础架构，而入门计划则提供更加灵活的选项。2017 年 7 月，Hyperledger 发布了 Hyperledger Fabric 1.0 生产就绪版，它由来自 28 个组织的 159 名开发人员共同开发完成，由企业社区构建，服务于企业社区。Hyperledger 的技术指导委员会根据企业采用需求，推动社区积极参与并做出贡献，支持生产网络的模块化、可扩展性和共识。

Hyperledger Fabric 提供核心功能，满足权限区块链网络的特定需求，网络的组织成员既有大型企业，也有小型企业。Hyperledger Fabric 的整个架构都具有模块化特点，让用户能够根据联盟的需求，交换密码、身份、共识算法、智能合约语言和其他方面的各类实施技术。Hyperledger Fabric 为用户构建分布式的业务网络提供强大的基础，使用户无须拼凑不同的解决方案即可完成构建。

（2）模块化

区块链网络必须能够根据企业和行业，融合各种全新的和现有的“可插拔式”功能。因此，Hyperledger Fabric 采用模块化设计，以适应新功能的增加，打造面向未来的网络。Hyperledger Fabric 的重要功能都具有模块化特点。

1）共识：支持任意基于投票的共识算法，满足崩溃容错（crash fault tolerance）为了满足拜占庭容错（byzantine fault tolerance）的要求。目前基于实施配套 Apache Kafka，同时还在开发搭配其他选项的产品，比如基于 Raft 和基于 BFT-SMaRt 的选项。

2）数据库：目前提供的数据库选项包括 LevelDB 和 CouchDB，其他选项还在开发中。

3）成员服务：目前基于 Public Key Infrastructure 实施，即将推出基于零知识证明（Zero-Knowledge Proof）的实现。

借助 Hyperledger Fabric 的模块化设计，IBM Blockchain Platform 能够利用行业领先的安全实践，服务于生产就绪型网络。

（3）可扩展性

随着各行各业完成初始的探索和概念验证，企业需要可扩展的解决方案。Hyperledger Fabric 能够支持不断增长的业务网络，动态地增加参与者，并支持与日俱增的交易处理需求。可扩展性在很多方面取决于共识、成员或安全配置。模块化平台能够配置网络，从而达到所需的吞吐量规模。Hyperledger Fabric 能够进行扩展以支持吞吐量，根据不同的用例，满足企业需求。目前的网络每秒能够处理数千笔交易。

可扩展性的意义不仅体现在吞吐量上。网络的增长还要求新参与者能够便捷地加入网络，并在网络上开展交易。Hyperledger Fabric 将参与者分为背书人和提交人两种角色。这意味着，只想要账本副本的参与者可以作为提交人角色加入网络，提交人角色无须承担交易

仅供非商业用途或交流学习使用

背书的压力，即可更新账本副本。

最后，Hyperledger Fabric 引入了“渠道”理念，让参与者能够机密地完成交易，并加入多个有特定业务合作伙伴的渠道。通过利用治理和网络配置工具，IBM Blockchain Platform 进一步增强了这些功能。分布式的业务网络需要一个能够动态增加参与者、资产和交易的平台。

（4）共识

对于每个区块链协议的安全性、可扩展性和成熟度来说，清晰定义并实施的共识算法可能是最重要的功能。选择适当的共识算法对于在分布式的业务网络中支持互相信任至关重要。

如上所述，Hyperledger Fabric 中的共识具有可插拔的特点，以满足特定的企业用例需求。比如，安全需求相对有限的开发网络最好采用 SOLO 共识模型，从而用一个节点验证

所有交易；生产网络可能更需要崩溃容错和拜占庭容错共识算法。Hyperledger Fabric 支持上述两种模式。

目前，Hyperledger Fabric 支持许可型网络中基于共识算法的投票。投票和许可的结合让网络运行时的性能远远超过了许多公有拜占庭容错网络。没有未知角色，意味着我们不需要麻烦的共识算法。Hyperledger Fabric 提供开箱即用的 Apache Kafka，并支持崩溃容错。因此，当出现部分网络崩溃时，网络依然能够运转。

其他共识算法包括 BFT-SMaRt 和 SBFT（简化的拜占庭容错），以容忍共识中的恶意行动。Hyperledger 详细比较了不同的 Hyperledger Frameworks，包括 Hyperledger Fabric，并发布了相关信息。

（5）高度安全的基础架构

如上所述，基础架构的选择与服务计划的选择息息相关。IBM Blockchain Platform Enterprise 和 Enterprise+ 计划借助 LinuxOne Emperor，利用行业领先的安全性，确保所有代码和数据始终处于加密状态，经过篡改的虚拟机不会启动，不会出现管理员访问或特权访问。代码在 IBM Secured Services Containers（SSC）中执行，以保障账本的安全性。IBM Secured Services Containers 能够确保：

1）租户彼此隔离。

2）消除特权访问，避免出现内部攻击或凭证泄密。

3）数据密钥是私有的，即使法院传票要求，IBM 也无法访问数据。

4）可信的 Boot Loading，实现防篡改的代码执行。

在硬件安全模块方面，IBM Blockchain Platform 满足最高的 FIPS 140-2 Level 4 标准。

此外，IBM Blockchain Platform “始终在线”的设计能够在运行时更新网络，并基于全球最快的 Linux 计算优化了性能。以上每一项功能背后都有 IBM 深厚的 Hyperledger Fabric 专业技能的支持，IBM 全年全天候 24 × 7 提供直接融入控制台的区块链技术支持。IBM 加入了特定的工具和功能，让网络运行变得更轻松。其中包括：

1）仪表板，监控并管理网络上的资源。

仅供非商业用途或交流学习使用

2）生命周期管理，无须中断网络，即可无缝升级整个代码堆栈。

3）全天候 24×7 技术支持，集成至门户。

4）强化的安全堆栈，无特权访问，无恶意软件，防篡改，100% 硬盘加密，以及保护硬件安全模块密钥。

（6）网络运行

IBM Blockchain Platform 支持创始人通过一个简单的用户界面，启动、邀请和配置网络。启动网络时，需创建 3 个订购对等节点，2 个认证中心。这为创始人提供了一个现成的基础，创建自己的业务网络。然后，创始人可以利用任意数量的对等节点，邀请其他参与者加入网络。参与者将收到邮件邀请通知，从而轻松加入网络。

网络运行用户界面还支持创始人配置核心网络组件，比如身份验证和渠道创建。这有助于确保只有获得许可的用户能够访问网络，机密交易将通过渠道完成。

（7）业务运营

IBM Blockchain Platform 提供用户界面，从而在活跃的区块链网络中支持业务运营，无需放弃网络或中断运营即可完成更新。

智能合约能够自动交换信息和资产，是区块链网络的核心功能。IBM Blockchain Platform 用户能够通过单一用户界面，在网络中轻松部署和升级智能合约。

此外，用户还能编辑渠道的策略，该渠道负责管控共识。

（8）运营监控

随着交易和参与者的与日俱增，用户需要监控网络上的活动。IBM Blockchain Platform 提供 Network Traffic Dashboard 和 Network Health Monitor。通过这些仪表板，用户能够主动调整网络运营，清晰定义网络中的资源使用情况。

8.3.4 微软区块链服务

1. 简介

Azure Blockchain Workbench 是微软 Azure 服务和功能的集合，旨在帮助创建和部署区块链应用程序，以便与其他组织共享业务流程和数据。Azure Blockchain Workbench 可提供用于创建区块链应用程序的基础结构基架，使开发者能够专注于创建业务逻辑和智能协定。此外，借助 Blockchain Workbench，通过集成多种 Azure 服务和功能，帮助自动执行常见开发任务，可更轻松地创建区块链应用程序。

2. 架构

Azure Blockchain Workbench 使用多个 Azure 组件提供解决方案，从而简化区块链应用程序的开发。可以使用 Azure Marketplace 中的解决方案模板部署 Blockchain Workbench。该模板可让用户选择要连同 Blockchain Workbench 一起部署的模块和组件，例如区块链堆栈、客户端应用程序类型以及 IoT 集成支持。部署后，Blockchain Workbench 会提供对 Web 应用、iOS 应用和 Android 应用的访问权限。Azure Blockchain Workbench 架构图如图 8-3 所示。

仅供非商业用途或交流学习使用

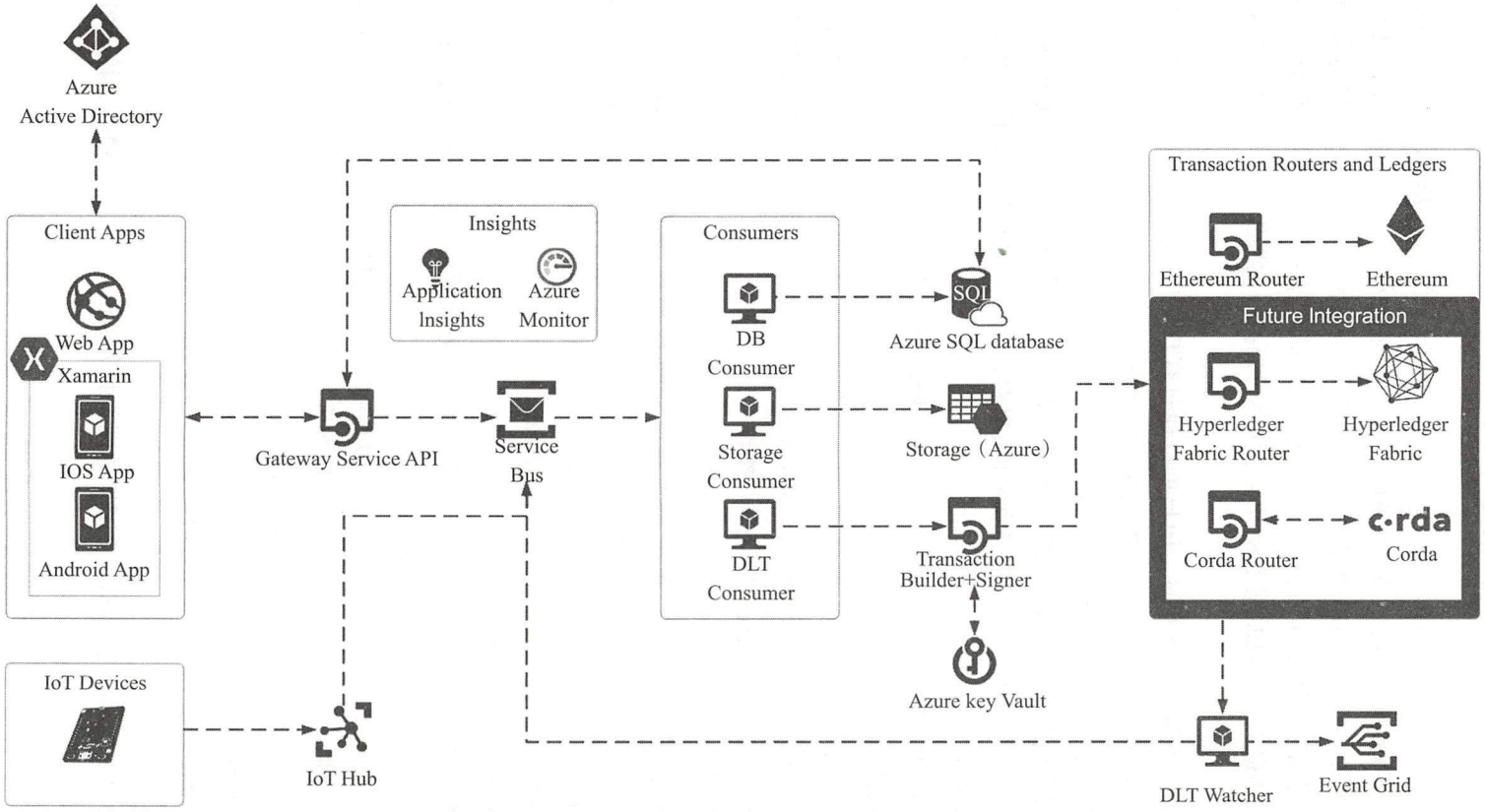


图 8-3　Azure 区块链工作台架构

仅供非商业用途或交流学习使用

仅供非商业用途或交流学习使用

（1）标识和身份验证

使用 Blockchain Workbench，联盟可以通过 Azure Active Directory（Azure AD）关联其企业标识。Workbench 使用 Azure AD 中存储的企业标识为链中标识生成新用户账户。标识映射简化了客户端 API 和应用程序的身份验证登录，并使用组织的身份验证策略。Workbench 还提供相应的功能用于将企业标识关联到给定智能合同中的特定角色。此外，Workbench 还提供一个机制用于标识这些角色可以执行的操作，以及何时可以执行这些操作。

部署 Blockchain Workbench 后，用户可以通过客户端应用程序、基于 REST 的客户端 API 或消息传递 API，与 Blockchain Workbench 交互。在任何情况下，都必须通过 Azure AD 或特定于设备的凭据对交互进行身份验证。

用户通过向参与者的电子邮件地址发送电子邮件邀请，将自己的标识关联到联盟 Azure AD。用户登录时，系统会使用名称、密码和策略对这些用户进行身份验证。例如，使用其组织的双重身份验证。

Azure AD 用于管理有权访问 Blockchain Workbench 的所有用户。连接到智能合同的每台设备也与 Azure AD 相关联。

Azure AD 还用于将用户分配到特殊的管理员组。与管理员组关联的用户将在 Blockchain Workbench 获得管理员权限 / 操作权限，例如，部署合同，以及向用户授予合同访问权限。此组外部的用户无权访问管理员操作。

（2）客户端应用程序

Workbench 为可用于验证、测试和查看区块链应用程序的 Web 和移动应用（iOS、Android）提供自动生成的客户端应用程序。应用程序接口根据智能合同元数据动态生成，并可适应任何用例。客户端应用程序向 Blockchain Workbench 生成的完整区块链应用程序提供面向用户的前端。客户端应用程序通过 Azure AD 对用户进行身份验证，然后提供根据智能合同业务上下文定制的用户体验。用户体验可让获得授权的个人创建新的智能合同实例，然后提供所需的功能，用于在智能合同表示的业务流程中的相应阶段执行特定类型的事务。

在 Web 应用程序中，获得授权的用户可以访问管理员控制台。控制台可供 Azure AD 管理员组中的用户使用，并提供以下功能的访问权限：

1）为常见方案（例如资产转让方案）部署 Microsoft 提供的智能合同。

2）上传并部署用户自己的智能合同。

3）在特定角色的上下文中为用户分配智能合同的访问权限。

（3）网关服务 API

Blockchain Workbench 包括基于 REST 的网关服务 API。写入区块链时，该 API 会生成消息并将其传送到事件中转站。当 API 请求数据时，会向链外 SQL 数据库发送查询。SQL 数据库包含链中数据和元数据的副本，这些数据提供受支持智能合同的上下文和配置信息。

仅供非商业用途或交流学习使用

查询以合同元数据指定的格式从链外副本返回所需的数据。

开发人员可以访问网关服务 API，来生成或集成区块链解决方案，而无需依赖 Blockchain Workbench 客户端应用。

（4）传入消息的消息中转站

想要直接向 Blockchain Workbench 发送消息的开发人员可将消息直接发送到服务总线。例如，可对系统到系统的集成或 IoT 设备使用消息 API。

（5）下游使用者的消息中转站

在应用程序的生命周期内会发生事件。事件可由网关 API 触发，或在账本中触发。事件通知可以根据事件启动下游代码。

Blockchain Workbench 自动部署两种类型的事件使用者。区块链事件触发其中的一种使用者来填充链外 SQL 存储。另一种使用者用于捕获 API 生成的事件的元数据，这些元数据与文档的上传和存储相关。

（6）消息使用者

消息使用者从服务总线提取消息。消息使用者的底层事件模型允许其他服务和系统的扩展。例如，可以添加 CosmosDB 填充支持，或使用 Azure 流分析评估消息。

下面介绍 Blockchain Workbench 中包含的消息使用者。

1）分布式账本使用者。分布式账本技术消息包含要写入区块链的事务的元数据。使用者检索消息，并将数据推送到事务生成器、签名器和路由器。

2）数据库使用者。数据库使用者从服务总线提取消息，并将数据推送到附加的数据库，如 SQL 数据库。

3）存储使用者。存储使用者从服务总线提取消息，并将数据推送到附加的存储。例如，在 Azure 存储中存储经过哈希处理的文档。

（7）事务生成器和签名器

如果需要将入站消息中转站中的消息写入区块链，DLT 使用者会处理此操作。DLT 使用者是一个服务，它会检索包含需要执行的事务的元数据的消息，然后将信息发送到事务生成器和签名器。事务生成器和签名器根据数据和所需的区块链目标汇编区块链事务。汇编后，事务将被签名。私钥存储在 Azure Key Vault 中。

Blockchain Workbench 从 Key Vault 检索相应的私钥，并对 Key Vault 外部的事务签名。签名后，事务将发送到事务路由器和账本。

（8）事务路由器和账本

事务路由器和账本提取已签名的事务，并将其路由到相应的区块链。目前，Blockchain Workbench 支持将以太坊用作其目标区块链。

（9）DLT 观察程序

分布式账本技术观察程序监视已附加到 Blockchain Workbench 的区块链上发生的事件。事件反映个人和系统相关的信息，如，新合同实例的创建、事务执行和状态更改等。系统会

仅供非商业用途或交流学习使用

捕获事件并将其发送到出站消息中转站，使其可供下游使用者使用。如，SQL 使用者会监视事件、使用事件，并在 SQL 数据库中填充包含的值。使用复制可在链外存储中重新创建链中数据的副本。

（10）Azure SQL 数据库

附加到 Blockchain Workbench 的 Azure SQL 数据库存储合同定义、配置元数据，以及区块链中存储的数据的副本（可通过 SQL 访问）。直接访问数据库即可轻松查询、可视化或分析这些数据。开发人员和其他用户可以使用该数据库进行报告、分析，或进行其他以数据为中心的集成。例如，用户可以使用 Power BI 将事务数据可视化。

此链外存储可让企业组织查询 SQL 中的数据，但不允许查询区块链账本中的数据。此外，通过将区块链技术堆栈不可知的架构标准化，链外存储可让用户跨项目、方案和组织重复使用报告和其他项目。

（11）Azure 存储

Azure 存储用于存储合同以及与合同关联的元数据。

从采购订单和提单，到新闻和医疗成像使用的图像，再到源自连拍摄像（包括治安监控摄像头）的视频和大型动画，文档在许多以区块链为中心的方案中发挥作用。文档不适合直接放入区块链。

Blockchain Workbench 支持使用区块链业务逻辑添加文档或其他媒体内容的功能。文档或媒体内容的哈希存储在区块链中，实际文档或媒体内容存储在 Azure 存储中。关联的事务信息将传送到入站消息中转站，然后经过打包、签名，最后路由到区块链。此过程会触发事件，而事件通过出站消息中转站共享。SQL 数据库使用此信息，并将其发送到数据库供以后查询。下游系统也可以使用这些事件，以对其进行相应的处理。

（12）监视

Workbench 使用 Application Insights 和 Azure Monitor 提供应用程序日志记录。Application Insights 用于存储来自 Blockchain Workbench 的所有记录信息，其中包括错误、警告和成功操作。开发人员可以使用 Application Insights 调试 Blockchain Workbench 问题。Azure Monitor 提供有关区块链网络运行状况的信息。

8.3.5 小结

区块链网络是一种通过技术实现的去信任的价值网络，在金融、物流、物联网等领域有广阔的应用前景，存在巨大的市场。很多云平台都在积极拥抱区块链技术，结合自身的云存储、云计算技术和其他 BCaaS 产品，选用不同的区块链技术，推出自身的区块链服务，并已经实现了商用，比如 IBM、腾讯、微软等。另外一些云平台也正在积极布局，即将推出区块链服务，比如 Oracle Cloud、百度云等。各 BCaaS 特点比较如表 8-2 所示。

仅供非商业用途或交流学习使用

表 8-2　各 BCaaS 特点比较

名称	区块链技术	自动部署	是否自主研发	服务支持	商用案例
IBM Blockchain Platform	Hyperledger Fabric	支持	否	强	有
TBCaaS	Hyperledger Fabric/TrustSQL（计划中）/BCOS（计划中）/Corda	支持	TrustSQL 自主研发	强	无
Azure Blockchain Workbench	Hyperledger Fabric/Ethereum/Corda 等	支持	否	一般	有

8.4　GDPR 对区块链的影响

大多数平台都有中央服务器来存储它们的数据及用户的个人信息。这些平台可以随意获取信息，并可以随心所欲地做任何事情，如，可以决定是否出售它、修改它，或将它留在那里直到永远。但所有这些将从 2018 年 5 月 25 日开始发生变化，欧盟的《通用数据保护条例》(以下简称《条例》)(GDPR) 将全面发挥作用。该条例包括 91 个条文，共计 204 页，经过两年的讨论后开始生效。《条例》的通过意味着欧盟对个人信息保护及其监管达到了前所未有的高度，可称得上史上最严格的数据保护条例。GDPR 的目的是加强对欧盟所有个人的数据保护，并为他们提供对自己数据的权利和控制。《条例》定义了数据三方（数据所有者、数据控制者、数据处理者）的责任和权限。

1. 处理数据须有合法理由

理由包括：数据主体的同意、履行合同需要、履行法定义务的需要，以及数据控制者的合法利益等。以下对数据主体的同意、数据控制者的合法利益以及敏感数据的处理等重点条款进行解释。

（1）数据主体的同意

《条例》对于数据主体的同意的有效标准相比以前的《欧洲数据保护指令》严格很多。虽然《条例》并没有明确禁止“推定同意”模式（敏感数据处理、数据画像活动除外），但在实践中通过推定方式获得用户同意将很难被认为是有效合法的。也就是说，当前实践中普遍存在的通过冗长晦涩的隐私政策来获取用户同意，或者让用户在签订业务协议时通过“勾选”方式做出一揽子授权的方式将失去合法性。业界普遍认为，《条例》关于有效合法同意的严格规定，使得用户的同意不会像现在这样被轻易获得。更重要的是，《条例》赋予了数据主体可以随时撤回同意的权利。数据控制者应当明确告知用户现有该权利，并为用户行使该权利提供便利。

（2）数据控制者的合法利益

《条例》赋予了数据主体对于营销活动的绝对反对权。换言之，数据控制者可以以营销为目的使用用户个人数据，一旦用户提出反对，数据控制者必须立即停止使用。除此之外，将数据控制者的合法利益作为数据处理的合法理由的情形在实践中非常有限。数据控制者必

仅供非商业用途或交流学习使用

须能够证明，其合法的利益显著高于数据主体的个人权利和自由（《条例》第 6 条）。

（3）敏感数据的处理

敏感的个人数据包括：能够表明个人的种族、政治倾向、宗教和哲学信仰以及关于个人健康的数据。在敏感数据类型中，《条例》还明确加入了基因数据和生物数据，这类数据的处理能够唯一地识别出特定个人（《条例》第 9 条）。

2. 数据主体权利

《条例》对数据主体的权利的规定细致入微，为个人有效行使权利提供了坚实的法律保障。

（1）知情权

《条例》规定数据控制者必须以清楚、简单、明了的方式向个人说明其个人数据是如何被收集处理的。可以想见的是，当前企业普遍应用的隐私政策必须进行大幅度改变，才能满足合规要求。

（2）访问权

数据控制者应当为用户实现该权利提供相应的流程，如果该请求是以电子形式提出的，则也应当以电子形式将数据提供给个人。控制者不能基于提供该服务而收费，除非数据主体的请求明显过量，超过负担（《条例》第 15 条）。

（3）拒绝权

对于两种情形，数据主体享有绝对的拒绝权：始终有权随时拒绝数据控制者基于其合法利益处理个人数据；始终有权拒绝基于个人数据的市场营销行为。数据控制者不仅要删除自己所控制的数据，还要求数据控制者负责对其公开传播的数据，要通知其他第三方停止利用并删除。这是对传统“删除权”的扩展。

总体看来，《条例》对于数据主体权利的补充完善，不仅极大增强了数据主体对于个人数据的控制能力，也对企业如何保障实现数据主体的权利提出了具体的要求，对企业的制度建设、措施配置、业务流程乃至 IT 系统设计产生直接影响。

3. 数据控制者的问责

《条例》大大简化了企业日常的合规负担，特别是废除了目前各成员国关于数据处理及境外转移的许可或者备案程序。但是取而代之的是要求企业在内部建立完善的问责机制，以实现《条例》规定的真正落地。特别是，《条例》旨在对个人数据处理中的个人权利和自由提供充分的尊重和保障，因此，对于数据控制者和处理者的约束规范十分严格。欧盟数据保护机构第 29 条工作组已经将制定相关细则列为工作优先项。

4. 数据处理者的问责

对于数据处理者而言，《条例》带来了重大变化。1995 年版的《指令》主要适用于数据控制者。数据处理者主要通过合同的方式承担数据保护责任。然而《条例》对于数据控制者、数据处理者在大多数情况下提出了相同的要求，例如数据处理者也承担对数据的安全保

仅供非商业用途或交流学习使用

障义务，应在管理措施、技术上采取必要的措施，包括指定 DPO、在发生数据泄露事故时及时报告数据控制者等。

《条例》中对数据处理者构建的一系列规范要求，将对当前的云计算生态体系带来重大影响。按照《条例》，数据控制者和数据处理者之间的合同在很多情形下需要重新谈判达成。特别是由于《条例》使数据处理者大大增加了合规风险，二者合同中关于安全保障措施、风险管理以及服务的价格都会受到影响。《条例》还从监管、数据流动、数据画像等多方面做了规定，在 GDPR 之下，权力将完全归还给数据所有者。某社交网站以及其他公司向第三方出售用户个人数据获得巨额财富的时代将结束。欧盟公民将拥有个人数据的专有权，并可要求从拥有该数据的公司的数据库中彻底清除个人信息。GDPR 法规的出台肯定是一件好事，因为它让数据所有者完全掌控自己的个人数据，并将“理智”带回互联网。目前热门的区块链平台，包括公链（如比特币或以太坊）和企业联盟链（如 Corda、Fabric 等）也同样需要遵守 GDPR 的规定。我们需要思考一个问题：目前这一代的区块链是否符合 GDPR 标准？至少存在以下挑战。

（1）数据删除上的挑战

GDPR 的数据删除规定，通常被称为“被遗忘权”。GDPR 第 17 条所述，被遗忘权要求个人数据必须被新定义下的“数据控制器”删除。这些数据也限制被进一步传播给第三方实体。区块链技术目前是最安全、最透明和不可变的数据存储系统，这是众所周知的事实。存储在区块链中的数据不能编辑、出售给第三方或删除。实质上，一旦信息进入区块链，就意味着永远保存在那里。与典型事务数据库的“CRUD”（创建、读取、更新、删除）体系结构形成对照的是，典型的区块链数据库只有创建、读取和更新，而没有删除，这就说明对基于区块链系统的公司来说，几乎不可能达到 GDPR 理想中的那样。从区块链技术特征分析，修改区块链上的数据非常困难。如果要删除或修改区块链中的数据以符合 GDPR 的修改权利或“被遗忘权”，除了需要改数据之外还包含数据和所有数据的块的散列后续块。

（2）数据属地上的挑战

GDPR 的适用范围取决于属地因素，要么机构的成立地在欧盟，要么利用欧盟境内的设备进行个人数据的处理活动（仅仅是传输通道除外），这就说明只要在规定范围内的数据都受到规定的保护。我们知道区块链的分布式特点，其节点有的在欧洲，也完全可能在欧洲以外有区块链的节点，区块链数据的去中心化特性实际上使得所有“个人数据”一旦离开欧洲即定为非法。

面对 GDPR 史上最强的用户数据保护条例，区块链的出路在哪里呢？

（1）区块链技术创新，适应 GDPR 的合规管理

预计将会有越来越多围绕企业技术的解决方案创新，以促进 GDPR 合规和消费者数据隐私。APEX Network 是一家创新的区块链平台公司，该公司围绕消费者数据管理和隐私提供的解决方案采用了一种链上和离链混合的方法。APEX Network 为企业提供智能合约功能，内置消费者数据管理协议，称为 ATDM（APEX Transactional Data Management），可嵌

仅供非商业用途或交流学习使用

入现有应用程序和客户接触点，以促进消费者的同意、身份管理、访问控制和数据加密。消费者可以跨应用程序管理自己的数据和配置文件，并将数据安全匿名地存储在云中。这是由ATDM 通过将区块链中的数据记录发送到 DHT（分布式哈希表）云中完成的，消费者对数据使用他们自己的私钥进行加密。该公司进一步计划让 APEX Network 的投资者和社区参与托管 DHT 云节点，在这些节点中，通过数据交易费用，极有可能实现激励。该解决方案已经在全球范围内试点企业用户，以解决与消费者数据隐私、信任和客户关系管理有关的一系列问题。APEX Network 这类的企业采用去中心化技术，使消费者能够保留对数据的所有权和控制权，同时，它可以帮助企业轻松嵌入协议，以提供对 GDPR 的一系列合规性。

（2）以区块链技术促进 GDPR

GDPR 是在一个中心化数据的前区块链世界中形成的，而区块链技术以尽可能多的去中心化性质而闻名。在欧盟《条例》的框架下，新一代的法律和技术正在将监管机构和创新者聚集在一起。人们只能希望欧洲的政策制定者能够洞悉区块链技术到底能在多大程度上帮助GDPR。使用区块链作为一种监管技术本身很有潜力，具体来说，就数据保护和数据主权的概念而言，已经有不少的提议，即区块链技术如何能让个人对自己的数据有更多的控制权。GDPR 和区块链之间是存在共同点的，GDPR 可能需要更多的时间来开发和理解区块链是如何设计的，以帮助实现 GDPR 提出的高标准的数据隐私。我们需要探讨未来区块链如何在元级层面上实现兼容，如果设计合理，区块链可以实现 GDPR 的潜在目标，即让数据主体对其数据进行更多的控制。这是监管机构在区块链方面面临的根本问题：它们既是威胁，也可能是数据保护视角下的机遇。GDPR 明确追求的目标是让个人对自己的数据拥有更多的控制权，尽管这些区块链在起草过程中绝对不是监管机构想要的那样，但最终可能会实现这一目标。

8.5 区块链面临的挑战

8.5.1 待解决的四大难题

区块链行业正处于飞速发展的时期，在技术和行业高速发展的阶段，也显现出了不少问题。其中有些问题对区块链行业的发展会产生非常不利的影响。

1. 尚未建立统一的标准

区块链到底是什么，目前业界尚没有一个统一的清晰的概念。没有统一清晰的概念界定，又缺少权威的机构对区块链产品进行评定，这极易造成在涉及区块链的项目谈判、实施过程中出现问题，更谈不上区块链的大规模推广和应用。市场上已有的区块链应用也是“鱼龙混杂”，无法有效评价产品质量。

区块链亟须建立一套统一的标准规范来界定其内涵和外延，并说明评判的方法，从而引导市场健康发展。然而区块链技术仍在不断创新变化，应用场景也在探索之中，过早的标

仅供非商业用途或交流学习使用

准化会限制区块链技术的创新和行业的发展。因此，为适应目前区块链行业的发展阶段，区块链标准化工作应从满足用户的真正需求角度出发，以测试某个区块链系统对用户需求的匹配度为原则，开展功能和性能测试的“黑盒”标准化，而不是过早地对区块链技术进行规范。

2. 衍生市场混乱

处于炒作高峰期的区块链技术不仅受到社会大众的关注，而且存在着被不法分子所利用进行欺诈的情况。目前市场上出现了大量打着数字货币旗号进行传销、诈骗、非法集资的假数字货币。这些假数字货币利用门户网站、微博、微信公众号、贴吧等渠道进行宣传和招商等活动。甚至有些假数字货币还在虚拟货币交易平台上线交易和炒作。这些假数字货币，除了给广大投资者带来经济损失之外，也让区块链技术不明不白地背了黑锅，阻碍了区块链行业的正常有序发展。

2017 年 9 月 4 日，中国人民银行、中央网信办、工信部、工商总局、银监会、证监会和保监会联合发布了《关于防范代币发行融资风险的公告》，其中对代币发行融资活动进行了明确的定位，“本质上是一种未经批准非法公开融资的行为”。10 月 5 日，国务院办公厅就推进供应链创新与应用发布指导意见，其中提到了研究利用区块链、人工智能等新兴技术，建立基于供应链的信用评价机制。政府相关部门的一系列政策，非常及时地将区块链技术和金融活动区分开来，让人们充分认识到区块链技术和代币的区别，并为区块链技术发展明确了方向。

3. 不容忽视的安全

由于大量资本进入区块链行业，区块链技术在近期得到了快速发展。在技术快速发展的同时，技术的安全问题没有得到相应的关注。从原理上讲，区块链技术具有很高的安全性和可信性，然而在工程实现上和实际应用中，未必能达到期望的效果。

例如，在账本模式的账务类区块链系统中，私钥是用户身份的唯一凭证。而在实际业务中，尤其是面对普通消费者的业务中，需要将私钥和消费者的社会身份进行绑定，并且由区块链系统运营方来代替消费者保管私钥。这种情况下，密钥管理的安全问题对整个商业模式而言，已经成为一个非常重要的问题。而这个安全问题并不能通过区块链技术自身来解决，而是需要在区块链系统外部解决。

4. 监管难度空前

区块链技术采用“去中心化”的技术设计，避免了传统中心化经济系统结构中的诸多问题。但去中心化也意味着主体不明确，监管难以对主体进行有效控制。2017 年 5 月 12 日发生席卷全球的“勒索病毒”事件，犯罪分子以比特币作为交易赎金，导致对其身份的追查格外困难。

但正是由于区块链的消息同步、易接入、共享账本等特性，能够允许监管机构作为一个节点接入网络，获得最全、最及时的监管数据，避免了传统监管方式中数据造假的问题。

仅供非商业用途或交流学习使用

因此，政府对区块链应用的监管，很有可能不是采用传统的行政命令方式，而是让监管机构本身也参与系统当中。正确地应用区块链技术，可以增强政府的监管力量。尤其是使用智能合约等进行合规性审查，能够依托区块链自身的公开透明和自动化运行，有效降低审计的成本，显著缩小监管的需求和范围。

为了加快我国区块链行业的健康发展，让我国在全球参与的区块链技术竞赛中处于领先的地位，政府相关部门应该尽快开展以下四方面工作：一是稳步推进区块链标准化测试体系建设，二是鼓励开源区块链技术的发展，三是开展有效的区块链人才培养工作，四是加强区块链的监管和安全技术研究与实践。

8.5.2 性能问题及解决建议

1. 性能问题

在金融行业，每天都会产生大量的数据交易。如果采用区块链技术，那么就意味着每个人都有一份完整账本，并且有时需要追溯每一笔记录。因此随着时间推进，交易数据超大的时候，就会有性能问题，如第一次使用需要下载历史上所有交易记录才能正常工作，每次交易为了验证你确实拥有足够的钱而需要追溯历史每一笔交易来计算余额。

区块链的去中心化和分布式并不是同样的概念。在分布式网络中，每个节点都是一个大的计算任务的子集，节点可以根据自身能力分配算力，能者多劳，最终再汇总结果。假设总任务量是 W、网络节点数是 N 的话，那完成任务 W 的时间是 W/N。在这样的网络当中，节点越多整体网络的性能越好；而在区块链的去中心化网络当中，每个节点都需要独立的存储账本，完成共识并运行智能合约，并且整个交易必须等到每一个节点都完成交易并达成共识之后才能确认，因此整个网络的效率是 $W \times N$。区块链系统的瓶颈是网络当中算力最差的那个。因此对区块链应用，绝不能采用分布式那种随便拉几台机器，效率不行就增加机器的办法。当前的技术情况下，整个网络的总存储和计算能力取决于单个节点。甚至当网络中节点数过多时，可能会因为一致性的达成过程延迟而降低整个网络的性能。尤其在公有网络中，由于大量低质量处理节点的存在，问题将更明显。

决定区块链的性能的因素有很多种，如网络结构、加密算法、共识机制等，但最重要的还是交易是否可以被并行验证。如果交易可以被并行验证，那么未来就可以通过简单地增加 CPU 数量来提高吞吐量。基于 UTXO 系统的比特币可以很容易地对交易进行并行验证，因为 UTXO 之间是没有关联的，对任何一个 UTXO 的状态改变都可以独立进行且与顺序无关。而基于余额的账户系统则不那么容易实现并行，因为可能会同时发生多笔交易对同一个账户进行资产操作，需要进行一些额外的步骤来处理。举个例子，假设账户中的余额为 10 元，有两笔针对该账户的交易同时发生，第 1 笔交易在账户中 +5 元，而第 2 笔交易在账户中 -11 元。那么如果先执行第 1 笔交易，则两笔都能成功，最终余额为 4 元；如果先执行第 2 笔交易，那么它会因余额不足而失败，只有第 1 笔交易会成功，最终余额为 15 元。而对交易的并行验证起到决定性作用的，是智能合约是否具备状态持久化的能力。如果一组合

仅供非商业用途或交流学习使用

约都是无状态的，那么它们就可以按任意的顺序被执行，不会产生任何副作用；相反，如果合约可以对一组状态产生影响，那么按不同的顺序来执行合约产生的结果也会不同。举个例子，一个计算存款利息的合约，它具有两个子功能：存款和利息结算。假设账户中有 100 元，利息为 10%，现在同时发生了两笔交易，第 1 笔交易的内容是存入 100 元，第 2 笔交易的内容是结算利息。假如第 1 笔交易先执行，那么最终账户的余额是：（100+100）× 110%=220 元；如果第 2 笔交易先执行，那么账户余额将是：100*110%+100=210 元。由此可见，具备状态持久化能力的智能合约是顺序相关的，因此难以并发验证，特别是如果合约之间还可以相互调用的话，情况将会更加复杂。

2. 解决思路

如何提高交易的吞吐量，同时降低交易的确认延迟，是解决区块链性能问题的关键。区块链系统跟传统分布式系统不同，其处理性能在现有技术下，无法通过单纯增加节点数来进行扩展，实际上，在很大程度上取决于单个节点的处理能力。高性能、安全、稳定性、硬件辅助加解密能力，都将是考察节点性能的核心要素。一方面可以将单个节点采用高性能的处理硬件，同时设计优化的策略和算法，提高性能；另外一方面将大量高频的交易放到链外，只用区块链记录最终交易信息，如闪电网络等。类似的，侧链（side chain）、影子链（shadow chain）等的思路在当前阶段也有一定的借鉴意义。类似设计可以很容易地将交易性能提升 1 ～ 2 个数量级。此外，如果采用联盟链的方式，在一定的信任前提和利益约束下优化设计，也可以换来性能的提升。当前的各公链在并发处理能力、可伸缩性方面的不足，导致无法承载广泛的业务场景。为了解决这两方面的问题，目前业界从不同方向出发提出了各种解决方案：

（1）增加区块的大小

例如，在比特币中一个区块大小约有 1MB，SegWit2x 曾将区块大小提高到 2MB 以提高吞吐量，但 Bitcoin Core 最终出于对安全性和其他因素的考虑取消了 SegWit2x 硬分叉。由于计算性能及带宽的限制，区块大小在提升到一定程度后，矿工在短时间内难以完成全网广播，可能导致整个网络不能正常运行，因此这个方案仅能非常有限地提升吞吐量。

（2）链下交易

如比特币的闪电网络（Lightning Network）、以太坊的 Raiden Network 等。在这些方案中，用户可以提前支付一些以太坊或比特币作为押金，之后在链下与他人进行交易，交易结束后进行总的结算，最终将结算结果放在区块链上。这种方式可以大幅度提升系统的吞吐量，达到每秒上万甚至更高的交易数，但不足之处在于链下交易失去了开放性、透明性的优势，相当于成为客户端服务器的一个终端，并且没有那么多节点进行行为监督，也就淡化了去中心化的优势。

（3）代理人共识协议解决方案

即所有矿工通过权益证明或官方验证等方式选举出一定数量的代理人，选出的代理人就产生区块达成共识，之后广播给整个网络，从而达成整个网络的共识。正如上个问题说到

仅供非商业用途或交流学习使用

的那样，这个方案的好处是共识是在很小的团体内部达成的。因此速度较快。但相较比特币、以太坊即共识由成千上万节点共同决策而言，这一方案容易被质疑去中心化和安全性不足。此外，因为这一小部分的代理人节点可能是由同一个利益团体选出来的，他们是否能代表绝大多数人的利益尚待考证。

（4）分片技术

分片受传统数据库分片概念的启发，就像数据库被切分成几部分放置在不同的服务器上一样，在公有区块链中，交易被划归到不同的分片（shard）上同时进行处理，也就是说每个节点只处理整个网络中一小部分的交易，并且这个处理过程是与整个网络中的其他节点同时进行的。这就意味着，加入网络的节点越多，分片的数量也越多，整个网络能够同时处理的交易也越多。在所有的链上扩展解决方案中，分片技术是独一无二的，因为它带来的扩展是横向的，即网络吞吐量随着矿工节点数量的增加而增长，这是其他解决方案不具备的特性。正是因为分片的这种特性，再加上它是链上的、任意节点都可以加入的、去中心化的，它很可能成为推动区块链技术迅速普及的理想动力。

虽然可以通过一些技术手段（如索引）来缓解性能问题，但问题还是明显存在的，以下是解决性能问题的一些通用化建议。

（1）做好规划

对于区块链验证节点这类核心节点，不仅要选用性能和稳定性都好的高端服务器，同时还要尽量保证网络中的节点性能差距不要过大，不然性能差的节点会成为木桶原理中的短板，影响整个系统的性能。

（2）软硬结合

区块链平台技术还在不断演化，这中间必然会产生大量的优化，从安全性、健壮性和效率上都会有提升。

尽管区块链是开放的技术，但是它对基础架构平台的性能有相当高的要求。LinuxONE 通过内置的 CPACF 芯片进行 Hash 运算，可将哈希性能提升近 10 倍。SIMD 指令集将椭圆曲线加密的速度提升数十倍，同时大幅度节约 CPU 开销。根据近期的测试，LinuxONE 平台在某些场景的 TPS 可以达到 2000+，响应时间可以达到毫秒级，而其他平台则不到这个值的 1/4。

（3）平台调优

推荐使用 pbft batch 模式而不是 Single 模式，因为共识是串行执行，那么一次性处理多个请求显然要比每个请求都做一次共识要来得快，batch size 目前建议是 100。建议优化智能合约的运行环境，目前智能合约主要是在 Go 语言的运行环境中执行，可适当提高 GOGC 大小。如果用户的 CPU 支持 SIMD 指令集的话，不妨尝试用 SIMD 指令集实现椭圆加密算法的部分实现，实测可以最高提升 30 倍的算法性能。另外如果打开安全开关的话，建议采用证书批量发放的模式设置 Tcert Batch Size，可提高 CA 节点的证书方法效率。在实际测试场景中，CA 节点的压力是验证节点的 2 ～ 3 倍，建议多给 CA 节点分配资源。

仅供非商业用途或交流学习使用

8.5.3　安全问题及解决建议

区块链安全是一个系统工程，系统配置及用户权限、组件安全性、用户界面、网络入侵检测和防攻击能力等，都会影响最终区块链系统的安全性和可靠性。区块链系统在实际构建过程中，应当在满足用户要求的前提下，在安全性、系统构建成本以及易用性等方面取得一个合理的平衡。传统的安全问题如下：

1）区块链相关软件存在漏洞。包括 Parity 多重签名钱包漏洞、交易所安全漏洞等。

2）木马与病毒。日本 Coincheck 公司员工计算机中存在恶意软件，造成约 5.3 亿美元的损失。韩国 Bithumb 公司员工计算机被入侵，损失数十亿韩元。转账木马、加密货币木马会在用户要进行转账的时候，将要转账的目标替换成攻击者的账户。

3）假新闻与社工。2018 年 3 月，出现币安钓鱼事件。2018 年 4 月，MyEtherWallet 遭 DNS 劫持攻击，用户被劫持到假冒网站。

针对区块链的攻击面有：针对钱包的攻击（盗取私钥）、针对交易所的攻击（API 安全、提币、卖出或做空收割）、针对矿池的攻击（将挖矿收益账户改成黑客的）、针对区块链平台的攻击（粉尘攻击，即大量微小交易）、针对智能合约漏洞的攻击（比较常见）。

随着数字货币的产生，还出现了一些新的安全问题，比如：智能合约的安全（形式化证明）、51% 攻击，还有挖矿恶意软件。原本黑客控制大量“肉鸡”（被黑客远程控制的机器），黑客还会把黑掉的机器配置成挖矿机，利用它们的算力来挖数字货币赚钱。有的黑客还会专门在访问量高的网站页面或者博客植入挖矿脚本，用户一旦访问这样的网站，计算机的处理器性能就会瞬间被占满，成为一个挖矿节点，为黑客的矿池提供算力。

智能合约是区块链安全中最重要的一个风险点。从统计数据看，开源代码大约每 1000 行就含有一个安全漏洞，表现最好的 Linux kernel 2.6 版本的安全 bug 率为每 1000 行代码 0.127 个。而智能合约作为新生事物，对应的程序员没有经过严苛训练与考验，其代码可靠性会差很多。智能合约的安全机制包括：

1）代码定型与发布时的验证与检查。无论设计者是否愿意，每个发布的代码将接受自动规则验证检查，从而确保静态代码审查通过，那些典型的溢出漏洞规则将无处藏身。

2）节点在执行合约中的动态验证。该动态验证将涵盖本合约、关联合约的验证，并对执行过程中的状态进行审查，从而实现对各种执行漏洞进行弥补，即使黑客造出漏洞，各个合约执行者也会严密审视，并挂起正在执行的操作。

3）合约执行完毕的合理性判断。合约执行完毕的结果将通过一定的规则进行评判，同时引入人工智能，对合约执行的合理区间进行分析，从而决定最终的结果输出；例如对账目进行复式审查或从更高维度进行审查。

4）相关利益方的申诉机制与自动判决技术。在智能合约部署的节点上，每个节点都内置基于规则的判决机制以及人工智能审核机制，支持自动投票表决，从而保证有一定的机会挽回损失。

智能合约的安全机制可以采用以下几类技术来实现：

仅供非商业用途或交流学习使用

（1）基于规则知识库的语法检查

核心原理是将原始编码文件通过内置编译工具，将合约构建为一棵基于 BNF 范式基础上的抽象语法树（AST），通过该语法抽象树便可以对合约内容展开语法识别，进行简单的合约安全识别。目前建议按照递归下降分析的方法，对语法抽象树进行基于知识规则库的检查，从而确定是否存在安全隐患。虽然一般的智能合约描述均为图灵完备，抽象语法树可以表现为多样性，但很容易发现：安全的智能合约实际应该是一个典型的闭合自治描述，具备有限状态空间或确保能够检测终止的有限状态机。因此可以通过检测的语法抽象树的平衡和闭合性，确定智能合约是否具备基本安全性。

典型的例子包括：对所有的条件选择语句进行完备性补足，防止由于条件不完善导致合约执行缺陷；对所有 public 成员与函数进行引用对象分析，确定合约对外暴露的危险等级；进行交易步骤完备性检查，确定每个合约交易方的条件动作描述完备。

（2）基于语义分析的交易模型识别与安全检查

基于语法的安全检查规则仅能静态识别合约缺陷，而基于语义分析的交易模型识别与安全检查，则主要通过上下文相关审查，确定智能合约中不满足规则或者不安全的操作。目前支持的安全检查包括：

1）类型检查，具体包括检查合约中需要对外暴露的对象与方法，审查其动作的必要性以及潜在的缺陷。

2）控制流检查，具体包括检查合约中各种选择分支或者针对 Oracle 的处理是否完备，并确定合约被调用时，是否存在其他异常处理等。

3）一致性检查，具体包括同一个合约条件，出现在不同的选择组合中；各种分支出现组合覆盖等，避免由于分布式执行出现由于矿工调用顺序不同导致的合约异常。

通过上述静态语义分析，能够基本排除由于人为书写智能合约带来的各种表层的逻辑缺陷，但尚不能解决动态执行过程中出现的各种逻辑问题。这些问题包括：

1）书写代码不精确、不完备导致的合约组合条件情况处理的缺失。

2）个人合约设计目的与真实编写代码之间存在较多的差异。

3）由于合约执行采用分布式执行，各个节点对代码的执行顺序存在差异，导致当本合约出现异常时，其他合约能够调用或更改本合约的各种状态，出现各种非安全性问题。

（3）基于 AI 的形式验证的智能合约安全性检查

MATRIX 是该领域的一个商业化解决方案，MATRIX 的核心是人工智能辅助计算，各个层级上均内置 AI 能力，因此在合约验证上，采用基于 AI 辅助的形式验证，以及动态约束检查的方法解决上述安全问题。其核心思想包括：

1）利用模式匹配获得用户真实需求约束，基于语义分析形成的合规语法抽象树进行基础模式匹配，获得用户可能的交易基础模型。该方法能够以静态手段获得大部分语法抽象分支的局部匹配。MATRIX 根据具体的匹配度，确认候选模型或模型组合，从而根据模型添加交易约束与交易断言。

仅供非商业用途或交流学习使用

2）对静态语义分析形成的抽象树，按照 MATRIX 的 AI 引擎——贝叶斯分类器进行模型分类，确定树中的各段分支属于对应的类属。而在 MATRIX 中，针对每个交易类属，均具备对应的静态与动态约束。

3）根据模式匹配结果和人工智能分类结果，获得当前合约的全部静态与动态约束，基于该约束即可生成合约代码的断言，并基于该结果进行形式验证和动态验证。

4）对于模型匹配失败或者分类失败的合约，MATRIX 将提出不可靠安全告警，并在执行过程中进行更严苛的边界检查。

MATRIX 支持 Bytecode 级别的语义审查，核心是进行反汇编，生成语法抽象树，然后利用 AI 进行语法树匹配。

本节重点介绍了区块链，特别是联盟链所面临的各种问题及对应解决思路和方法，能够协助读者从头构建起一套适合自己应用场景的区块链业务服务。

区块链技术虽然还在不断演进中，存在这样那样的一些问题，但我们看来区块链技术的这些问题都是成长过程中的问题。社区也意识到并有各种创新项目试图解决这些问题。这些问题也并不影响区块链的存在意义与价值，区块链经济的前景极为壮阔。一种乐观的预测认为，到 2025 年之前，全球 GDP 总量的 10% 将利用区块链技术储存。

现在，区块链经济已经处于爆发前夜。金融行业在这方面的探索领先一筹，而其他行业的应用正在快速展开。区块链行业应用具有明显的效益，其显著优势在于优化业务流程、降低运营成本、提升协同效率，这个优势已经在金融服务、物联网、公共服务、社会公益和供应链管理等社会领域逐步体现出来。全球范围内我们会看到出现越来越多的区块链应用。区块链带来了效率提升和成本降低的技术手段，为经济社会的发展和治理提供新的思路。围绕区块链体系，能够创造出丰富的产品和服务，人们可以在相互无信任的情况下，无地域限制地进行大规模协作。由此，一个全新的经济时代将展现在公众面前。

仅供非商业用途或交流学习使用

参考文献

[1] 董宁，朱轩彤 . 区块链技术演进及产业应用展望 [J]. 信息安全研究 ,2017,3(3)：200-210.

[2] 唐文剑，吕雯，等 . 区块链将如何重新定义世界［M］. 北京：机械工业出版社，2016.

[3] 深蓝 .Fabric 部署 [EB/OL].http://www.cnblogs.com/studyzy/p/7237287.html.

[4] Fabric 官网公开资料 [EB/OL].https://hyperledger-fabric.readthedocs.io/en/release/.

[5] Fabric 官网公开资料 [EB/OL].https://github.com/hyperledger/fabric.

[6] Fabric 官网公开资料 [EB/OL].https://www.ibm.com/blockchain/hyperledger.html.

[7] 邹均，张海宁 . 区块链技术指南 [EB/OL].https://www.gitbook.com/book/yeasy/blockchain_guide/details.

[8] 闫树，卿苏德，魏凯 . 区块链在数据交易中的应用［J/OL］.http://www.ccsa.org.cn/article_new/show_article.php?categories_id=bcab4177-f8c4-ba8d-0161-44b1beea9882&article_id=expert_83c507d7-78f1-2627-45f5-5afcc8047e7e.

[9] greatandrew. 通用数据保护条例（GDPR）和区块链：威胁还是机遇 [EB/OL].http://www.8btc.com/gdpr-blockchain.

[10] Trent McConaghy. 区块链将彻底改变人工智能 [EB/OL].https://blog.csdn.net/R1uNW1W/article/details/79428353.

[11] 第三极区块链技术有限公司 .Fabric 中应用 CouchDB [EB] .

[12] 第三极区块链技术有限公司 .Fabric 安全应用和管理私钥证书 [EB] .

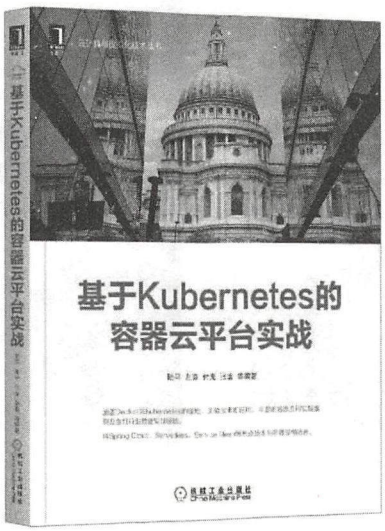
[13] 南京荣泽信息科技股份有限公司 . 区块链数据共享及服务实践 [EB] .

[14] 何宝宏 . 区块链 VS 大数据 [EB/OL].https://mp.weixin.qq.com/s?src=11&timestamp=1531624068&ver=999&signature=uF31-NFkQjoGq2JHJbESdM0ZuRRbImxuu7KIqggQe7umW6eapsAZ6aXzzPn-7er-uvp4Rs76nV6fa1fhKmJ5QbPxuCYWKFET3ruuPTkIjEVWkTtU3fq5tQKfzApr0PR0&new=1.

[15] 孟庆国 . 政务数据共享交换须先理顺“三权关系”[EB/OL].http://zb.cbdio.com/308/index.html?from=groupmessage.

仅供非商业用途或交流学习使用

推荐阅读



基于Kubernetes的容器云平台实战

书号：978-7-111-60814-1 作者：陆平 左奇 付光 张晗 等编著 定价：69.00元

这是一本深度讲解容器云领域关键技术及应用实践的著作，也是目前国内在容器云领域涵盖较广的技术专著。作者长期从事云计算、容器云等关键技术研究工作，有近十年的云计算、容器云平台一线研发经验。 本书内容由浅入深，以Docker技术基础介绍为开篇，详述了Kubernetes技术架构及原理，并提供了大量容器应用部署实例，有助于读者将理论与实践相结合，深入理解容器云平台；另外，本书将当前技术热点（如微服务、DevOps、Spring Cloud、Service Mesh等）与容器云相结合，并进行了深入的技术讲解，帮助读者了解最新技术前沿。

仅供非商业用途或交流学习使用

推荐阅读

机器学习与深度学习：通过C语言模拟

作者：[日]小高知宏 著 译者：申富饶 于傲 译 ISBN：978-7-111-59994-4 定价：59.00元

本书以深度学习为关键字讲述机器学习与深度学习的相关知识，对基本理论的讲述通俗易懂，不涉及复杂的数学理论，适用于对机器学习与深度学习感兴趣的初学者。当前机器学习的书籍一般只讲述理论，没有具体的程序实例。有些以实例为主的机器学习书籍则依赖于一些函数库或工具，无法理解其内部算法原理。本书没有使用任何外部函数库或工具，通过C语言程序来实现机器学习和深度学习算法，读者不太理解相关理论时，可以通过C语言程序代码来进行学习。

本书从强化学习、蚁群最优化方法、神经网络、深度学习等出发，分阶段介绍机器学习的各种算法，通过分析C语言程序代码，实际执行C语言程序，使读者能快速步入机器学习和深度学习殿堂。

自然语言处理与深度学习：通过C语言模拟

作者：[日]小高知宏 著 译者：申富饶 于傲 译 ISBN：978-7-111-58657-9 定价：49.00元

本书初步探索了将深度学习应用于自然语言处理的方法。概述了自然语言处理的一般概念，通过具体实例说明了如何提取自然语言文本的特征以及如何考虑上下文关系来生成文本。书中自然语言文本的特征提取是通过卷积神经网络来实现的，而根据上下文关系来生成文本则利用了循环神经网络。这两个网络是深度学习领域中常用的基础技术。

本书通过实现C语言程序来具体讲解自然语言处理与深度学习的相关技术。本书给出的程序都能在普通个人电脑上执行。通过实际执行这些C语言程序，确认其运行过程，并根据需要对程序进行修改，能够更深刻地理解自然语言处理与深度学习技术。

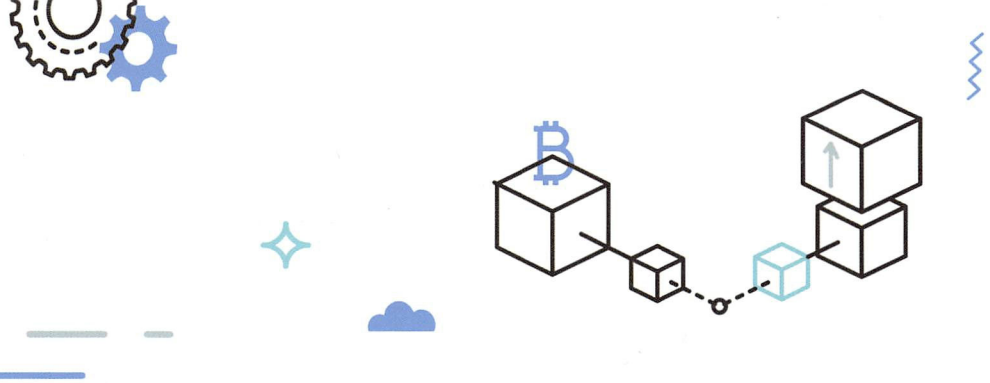
仅供非商业用途或交流学习使用

作者简介

陆平

博士，高级工程师，江苏省大数据存储及应用重点实验室主任，主要从事云计算、大数据、人工智能、区块链等方面的研究，是中国计算机学会 CCF 会员、服务计算专业委员会委员、CCF 大数据专家委员会委员、中国电子学会云计算专家委员会委员、中国人工智能学会图形图像专家委员会委员、中国通信学会评审专家、江苏省云计算工程技术中心主任、南京市软件企业联合会会长、南京市软件行业协会副理事长、东南大学“信息与电子专业国家级实验教学示范中心”教学指导委员会委员、东南大学产业教授、北京邮电大学和南京邮电大学兼职教授。主持和参与了国家科技重大专项、国家科技支撑计划、863 专项、发改委企业专项、物联网专项、江苏省科技成果转化项目等多项省部级科研课题，获得了省部级科技进步奖 10 多项，拥有 20 多项发明专利。撰写了《物联网能力开放与应用》《云计算中的大数据技术与应用》《云计算基础架构及关键应用》《OpenStack 系统架构设计实战》等著作，在国内外知名刊物上发表过多篇论文。

封面设计 · 陈潇然



全书可分为理论篇和实践篇两大部分，前三章注重理论，后五章注重实践，图文并茂、内容丰富、由浅入深、讲解全面，具有很强的可借鉴性。

本书重点从工程的角度完整地讲述一个区块链项目的背景、需求、项目方案和部署，全面地阐述了一个项目的功能设计、接口设计、流程设计，并从项目运营优化的角度提出了智能合约的设计和可视化二次开发、项目的可视化运营和部署的方案。结合不同领域的区块链典型案例的剖析，读者可对区块链技术到项目落地有一个较全面的认识。

本书最后对区块链未来发展进行了展望，从性能、安全等多维度对区块链跟其他技术的融合、区块链技术发展中面临的挑战、欧盟的“通用数据保护条例”（GDPR），以及区块链技术的相互促进等方面，进行了深入探讨。

上架指导：计算机/区块链

ISBN 978-7-111-60911-

9 787111 60911

定价：79.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn